Chapter 8
"XML-Enabled Applications"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter 3 “XML-Enabled Applications”

A synopsis of the book’s content

Information on where to buy this book

---

About the Author

Yuli Vasiliev is a software developer, freelance author, and a consultant currently specializing in open-source development, Oracle technologies, and service-oriented architecture (SOA). He has over 10 years of software development experience as well as several years of technical writing experience. He has written a series of technical articles for Oracle Technology Network (OTN) and Oracle Magazine.

---

PHP Oracle Web Development

Oracle Database is the premier commercial database available today, providing support for a wide range of features for professional developers. It's incomparable in terms of performance, reliability, and scalability. With the advent of Oracle Database XE, a lightweight edition of Oracle Database, you now have the option to use an Oracle database for free even in a final product. PHP is the most popular tool when it comes to building dynamic web applications. Unlike Oracle Database, PHP is an open-source product. The key reasons behind PHP's popularity are its ease of use, short development time, and high performance.

Even if you are new to PHP, getting started is pretty simple. When used in a complementary way, though, PHP and Oracle allow you to build high-performance, scalable, and reliable data-driven web applications with minimum effort.

*PHP Oracle Web Development: Data processing, Security, Caching, XML, Web Services, and AJAX* is a 100% practical book crammed full of easy-to-follow examples. The book provides all the tools a PHP/Oracle developer needs to take advantage of the winning combination. It addresses the needs of a wide spectrum of PHP/Oracle developers, placing the emphasis on the most up-to-date topics, such as new PHP and Oracle Database features, stored procedure programming, handling transactions, security, caching, web services, and AJAX.

What This Book Covers

*Chapter 1* gives an overview of the PHP and Oracle technologies, explaining why you might want to use PHP in conjunction with Oracle.

*Chapter 2* covers the basics of using the PHP OCI8 extension to interact with an Oracle database from PHP. It also briefly discusses some popular alternatives to the OCI8 extension to connect to Oracle from within PHP.

*Chapter 3* discusses how you can move data processing performed by your PHP/Oracle application into the database by using sophisticated SQL queries, stored PL/SQL subprograms, and database triggers.

*Chapter 4* discusses the various mechanisms that can be used to perform transactions with PHP and Oracle.

Chapter 5 examines the object-oriented approach to developing PHP/Oracle applications, as an efficient means to reduce the development time and complexity, and increase the maintainability and flexibility of your applications.

Chapter 6 looks at how to effectively use the security features of both PHP and Oracle together, examining the fundamental aspects of building a secure PHP/Oracle application.

Chapter 7 discusses how to effectively use caching mechanisms available in PHP and Oracle and provides several examples of caching in action.

Chapter 8 explains how to effectively use XML techniques and technologies available in PHP and Oracle when building XML-enabled PHP/Oracle applications.

Chapter 9 shows how to build a SOAP web service exposing the functionality of a PHP/Oracle application, using the PHP SOAP extension and Oracle XML technologies.

Chapter 10 explains how AJAX and some other client-side (browser-side) JavaScript technologies can be used along with the Oracle Database technologies as well as PHP features to improve the responsiveness of PHP/Oracle applications.

Appendix A discusses how to install and configure the PHP and Oracle software components required to follow the book’s examples.

Both PHP and Oracle provide comprehensive support for XML and XML-related technologies. Practically, this means you can perform any XML processing either with PHP or inside an Oracle database. While PHP allows you to construct and transform XML by using either PHP's XML extensions or PEAR XML packages, Oracle provides the Oracle XML DB, which has a wide set of XML features that can be used to efficiently store, retrieve, update, as well as transform XML data and generate it from relational data.

This chapter explains how to effectively use XML techniques and technologies available in PHP and Oracle when building XML-enabled PHP/Oracle applications. Specifically, you will see how to:

- Construct XML with the PHP DOM extension
- Navigate XML with XPath
- Transform XML with PHP XSL functions
- Generate XML from relational data with Oracle SQL/XML functions
- Store, retrieve, update, and transform XML with Oracle XML DB
- Validate XML documents against XML schemas
- Access XML stored in Oracle XML DB with standard internet protocols
- Query, construct, and transform XML with Oracle XQuery

**Processing XML in PHP/Oracle Applications**

As mentioned, there are two alternatives when it comes to performing XML processing in your PHP/Oracle application. You can perform any required XML processing using either PHP's XML extensions (or PEAR XML packages) or Oracle's XML features.
In the following sections, you will learn how to construct XML from relational data using the XML capabilities of both PHP and Oracle.

**Processing XML Data with PHP**

PHP provides three general extensions allowing you to work with XML. These extensions are listed in the following table:

<table>
<thead>
<tr>
<th>PHP extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML extension</td>
<td>The XML extension implements the SAX (Simple API for XML) approach to parsing and accessing XML content. The SAX parsing mechanism is memory efficient since it doesn't require the entire XML document to be stored in memory. This makes the SAX approach useful for certain type of operations on XML, for example, searching.</td>
</tr>
<tr>
<td>DOM extension</td>
<td>The DOM extension provides APIs for working with XML using DOM (Document Object Model). Unlike a SAX parser, a DOM parser builds an in-memory representation of an XML document, which in most cases makes performing modifying and updating operations more efficient.</td>
</tr>
<tr>
<td>SimpleXML extension</td>
<td>As its name implies, the SimpleXML extension provides the easiest way to work with XML. The SimpleXML approach allows you to access an XML document through its data structure representation and so can be especially useful when you simply need to read XML.</td>
</tr>
</tbody>
</table>

In practice, you should choose the extension that best suits the needs of your applications. For example, the XML extension implementing the SAX model can be very efficient when it comes to parsing large XML documents from which you only want to extract useful information. In contrast, the DOM extension comes in handy when you need to generate XML documents or modify existing ones. With the SimpleXML extension, XML documents are turned into data structures that can be then iterated like regular PHP arrays and objects, thus providing the most natural way for PHP developers to access data.

Since the Document Object Model (DOM) is best used for solving complex tasks, the following sections demonstrate how to use DOM extension APIs to generate, query, and manipulate XML documents in PHP.

Admittedly, the Document Object Model is widely used in web development. Web browsers, for example, use the DOM to represent web pages they display to the users. In Chapter 10 AJAX-Based Applications, you will learn techniques to access and manipulate the DOM tree of a web page sent to the browser by your application, thus allowing you to produce more interactive and responsive PHP/Oracle solutions.

Creating XML with the DOM PHP Extension

In fact, the PHP DOM extension is a set of classes that can be used to generate, access, and manipulate XML data. The DOM.php script defined in the following listing shows how to generate an XML document based on the result set retrieved from the database.

```php
<?php
    // File: DOM.php
    if (!$rsConnection = oci_connect('hr', 'hr', '//localhost/orcl')) {
        $err = oci_error();
        trigger_error('Could not establish a connection: ' . $err['message'], E_USER_ERROR);
    };
    $dept_id = 90;
    $query = "SELECT employee_id, last_name, salary FROM employees WHERE department_id = :deptid";
    $stmt = oci_parse($rsConnection,$query);
    oci_bind_by_name($stmt, ':deptid', $dept_id);
    if (!oci_execute($stmt)) {
        $err = oci_error($stmt);
        trigger_error('Query failed: ' . $err['message'], E_USER_ERROR);
    }
    $dom = new DOMDocument('1.0', 'UTF-8');
    $root = $dom->createElement('EMPLOYEES', '');
    $root = $dom->appendChild($root);
    while ($row = oci_fetch_assoc($stmt)) {
        $emp = $dom->createElement('EMPLOYEE', '');
        $emp = $root->appendChild($emp);
        $emp->setAttribute('id', $row['EMPLOYEE_ID']);
        $ename = $dom->createElement('ENAME', $row['LAST_NAME']);
        $ename = $emp->appendChild($ename);
        $salary = $dom->createElement('SALARY', $row['SALARY']);
        $salary = $emp->appendChild($salary);
    }
    echo $dom->saveXML();
    $dom->save("employees.xml");
?>
```

XML-Enabled Applications

To figure out what happens when you run the DOM.php script, let’s take a closer look at this code.

You start by connecting to the database as hr/hr. Then, you define a query, which, when issued, retrieves some information about the employees working in the department whose ID is 90.

After the query is executed, you create a new DOM document that will be used to wrap the retrieved result set in XML format. You start generating a new DOM document by creating the root element and then appending it to the DOM tree.

In the next step you create the nodes of the DOM document based on the data retrieved from the database. For this, you fetch the data from the result set in a loop, creating the document structure.

In this example, you simply display the generated XML document using the saveXML method of the DOMDocument object and then save it to disk with the save method to the same folder where the script source file resides. However, in a real-world situation, you probably would continue processing this XML document, producing a result XML document that could then, for example, be sent to a web service or published as an RSS feed.

When you run the DOM.php script discussed here, you probably will see the following string in your browser:

    King24000Kochhar17000De Haan17000

However, if you look at the source, you should see the following XML document:

    <?xml version="1.0" encoding="UTF-8"?>
    <EMPLOYEES>
    <EMPLOYEE id="100">
      <ENAME>King</ENAME>
      <SALARY>24000</SALARY>
    </EMPLOYEE>
    <EMPLOYEE id="101">
      <ENAME>Kochhar</ENAME>
      <SALARY>17000</SALARY>
    </EMPLOYEE>
    <EMPLOYEE id="102">
      <ENAME>De Haan</ENAME>
      <SALARY>17000</SALARY>
    </EMPLOYEE>
    </EMPLOYEES>

After running the `DOM.php` script, the `employees.xml` file containing the document shown in the listing should appear in the folder where the script source file resides.

**Querying a DOM Document with XPath**

One way to access the DOM tree in a `DOMDocument` object is through an associated `DOMXPath` object. Identifying a specific node or nodes within the DOM tree of a `DOMDocument` object with this approach involves use of appropriate XPath expressions passed to the `DOMXPath` object as parameters.

While the example in this section shows how XPath can be used in PHP, Oracle also has some SQL functions operating on XML, such as `existsNode`, `extractValue`, and `updateXML`, which take XPath-expression arguments.

The following script illustrates how to access XML content held in a `DOMDocument` object through the `DOMXPath` object associated with that `DOMDocument`.

```php
<?php
//File: XPath.php
$dom = new DomDocument();
$dom->load('employees.xml');
$xpath = new DOMXPath($dom);
$query = '//EMPLOYEE/SALARY[. > "15000"]';
$emps = $xpath->query($query);
print '<font face="Arial">';
print '<h3>Executive officers whose salaries > $15,000</h3>
<table border="1" cellpadding="5">
<th>Employee ID</th><th>Last Name</th><th>Salary</th>
foreach ($emps as $emp) {
print '<tr><td>'.$emp->parentNode->getAttribute('id').'</td>
print '<td>'.$emp->previousSibling->nodeValue.'</td>
print '<td>'.$emp->nodeValue.'</td></tr>
print '</table>';print '</font>;
?>
```

Unlike the preceding example where you generated an XML document from scratch, here you load it from a file, using the `load` method of the `DOMDocument` object. After the document is loaded, you create a new `DOMXPath` object, and associate it with the newly created `DOMDocument` object.

The XPath expression used in the above script is to be applied to the employees XML document loaded to DOMDocument object. You use this expression to identify all the SALARY nodes whose values exceed 15000, passing it to the DOMXPath's query method as the parameter.

For more information on XPath, you can refer to the W3C XML Path Language (XPath) Version 1.0 recommendation at http://www.w3.org/TR/xpath.

To iterate over the result set returned by the query issued within the script, you use the foreach construct. Since each row of the result set represents a SALARY node defined within its parent EMPLOYEE node, you access that parent node using the parentNode method of the DOMNode object representing the SALARY node being processed. However, to access the corresponding ENAME node you use the previousSibling method of the DOMNode object.

If you run the XPath.php script discussed here, your browser should display an HTML table representing the list of employees whose salaries exceed 15,000.

**Transforming and Processing XML with XSLT**

In the preceding example, you transform XML into HTML directly in your script, wrapping the data extracted from the XML document into appropriate HTML tags. Alternatively, you might perform an XSL (Extensible Stylesheet Language) transformation to get the same general results.

However, before you can use the XSL extension you have to enable it in your PHP installation.

In UNIX, you have to recompile PHP with the following flag:

    --with-xsl

In Windows, you have to uncomment the following line in the php.ini configuration file and then restart the Apache/PHP server:

    extension=php_xsl.dll

Once you have enabled the XSL extension, you can use XSL functions to transform XML into HTML or another XML or a variety of other formats. The following figure depicts the general steps performed by a PHP/Oracle application that generates an HTML page with PHP, based on the result set retrieved from the database.

Here is the explanation of the steps in the above figure:

- The script queries the database to retrieve the data that will be used to construct an XML document.
- The script generates the XML document using the PHP DOM extension, based on the data retrieved in Step 1.
- The script transforms the XML document generated in step 2 into HTML format with the PHP XSL extension.
- The script posts the HTML page generated in step 3 to the user's browser.

As you can see, most of the XML processing work in the above scenario is performed by the PHP engine on the web server rather than on the database server. So, this may be efficient in cases where the database server becomes a performance bottleneck in your system.

Using this scenario, you might transform the employees XML document shown in the Creating XML with the DOM PHP Extension section into HTML so that the result page looks like the following figure:

If you want to get the page shown in the above figure by applying an XSL transformation to the employees XML document, you first have to create an XSLT stylesheet describing the way the data is to be transformed.

The following employees.xsl stylesheet might be used to transform the employees XML document into HTML to get the page shown in the above figure.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Employees</title>
      </head>
      <body>
        <font face="Arial">
          <h2>List of employees from employees.xml</h2>
          <table border="1" cellspacing="0" cellpadding="5">
            <tr><th>EMPLOYEE ID</th><th>LAST NAME</th><th>salary</th></tr>
            <tr><td>100</td><td>King</td><td>24000</td></tr>
            <tr><td>101</td><td>Kochhar</td><td>17000</td></tr>
            <tr><td>102</td><td>De Hean</td><td>17000</td></tr>
          </table>
        </font>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

As you can see, the XSLT stylesheet shown in the listing is an XML document that contains elements and attributes defined in the XSLT namespace: http://www.w3.org/1999/XSL/Transform. Whereas the intent of these elements and attributes is to provide instructions to an XSLT processor, the HTML tags also presented in the stylesheet will be directly added to the resultant XML document.

While the employees.xsl stylesheet shown in the listing is designed to simply transform an employees XML document into HTML, you might create a more complicated stylesheet that would process XML data included in that XML document.

It is interesting to note that XSLT is not limited to transforming XML data—it also can be used to process XML. Sometimes, performing the XML processing with XSLT may be much easier than using DOM operations to do the same job. For example, with XSLT, to calculate the total of all the orders included in the document, you don't need to write the code that will iterate over all the elements representing that orders, as you would with the DOM approach. Instead, you might use the xsl:value-of select element with the sum function in your stylesheet to get the job done.

Turning back to the employees.xsl stylesheet, suppose you want to add another column to the resultant HTML table, say, BONUS whose values are calculated based on the values from the SALARY column. In that case, the fragment of the stylesheet responsible for generating the HTML table might be modified as follows:

```
<table border="1" cellspacing="0" cellpadding="5">
  <tr>
    <th><b>EMPLOYEE ID</b></th>
    <th><b>LAST NAME</b></th>
    <th><b>SALARY</b></th>
    <th><b>BONUS</b></th>
  </tr>
  <xsl:for-each select="EMPLOYEES">
    <xsl:for-each select="EMPLOYEE">
      <tr>
        <td><xsl:value-of select="@id"></td>
        <td><xsl:value-of select="ENAME"></td>
        <td><xsl:value-of select="SALARY"></td>
        <td><xsl:value-of select="SALARY*0.1"></td>
      </tr>
    </xsl:for-each>
  </xsl:for-each>
</table>
```

You might also want to calculate the average salary for the employees included in the employees XML document. To achieve this, you might further modify the employees.xsl stylesheet by adding the following XSLT construction immediately after the code shown above:

```
<p><b>Average salary is: </b><xsl:value-of
  select="format-number(sum(//SALARY) div
  count(//EMPLOYEE), '######0.00')"></p>
```

In this example, you sum the salaries of all employees included in the document with the \texttt{sum} function, and then divide the calculated sum by the number of employees obtained with the \texttt{count} function, thus getting the average salary formatted with the \texttt{format-number} function.

For more examples of XSLT stylesheets, you can refer to the W3C XSL Transformations (XSLT) Version 1.0 recommendation available at http://www.w3.org/TR/xslt.
Now that you have a grasp on how to create XSLT stylesheets to be used for transforming and processing XML data, it's time to see an XSL transformation in action.

The following listing contains a simple PHP script that performs an XSL transformation, applying the `employees.xsl` XSLT stylesheet defined earlier in this section to the employees XML document shown in the *Creating XML with the DOM PHP Extension* section. It is assumed that the `employees.xsl`, `employees.xml`, and the `XSLTrans.php` files reside in the same directory.

```php
<?php
// File: XSLTrans.php
$domxsl = new DOMDocument();
$domxsl->load('employees.xsl');
$proc = new XSLTProcessor;
$xsl = $proc->importStylesheet($domxsl);
$domxml = new DOMDocument();
$domxml->load('employees.xml');
$rslt = $proc->transformToXml($domxml);
print $rslt;
?>
```

After you have created a new DOM document, you load the XSL stylesheet discussed earlier in this section into that document. Next, you create a new `XSLTProcessor` object that is then used to perform an XSL transformation. However, before you can do this, you need to import the stylesheet into the newly created XSLT processor, and you also need to create a new DOM document and then load the XML document to be transformed.

For simplicity, in this example you do not query the database, nor do you generate a new XML document from scratch with the DOM functions. Instead, you load the existing document `employees.xml`, which was generated and saved to disk during the execution of the `DOM.php` script discussed in the *Creating XML with the DOM PHP Extension* section earlier in this chapter.

The XSL transformation performed in the above script transforms the employees XML document into an HTML page that you then send to the user's browser.

When you run the `XSLTrans.php` script defined in the above listing, the result should be something like the previous figure.
Performing XML Processing inside the Database

When building XML-enabled applications on top of Oracle, there are many advantages to performing the XML processing inside the database when compared to performing it on the client. The key advantages to perform XML processing inside the database are as follows:

- Benefiting from the XML-specific memory optimizations provided by Oracle XML DB
- Eliminating overhead associated with parsing XML documents
- Reducing overhead associated with I/O disk operations and network traffic between the Web server and database server

Moving XML processing to the database may be especially useful if you are dealing with large XML documents stored in the database. In that case, your application won't need to transfer a large amount of data between the database and web server when processing XML inside the database—only the final product is sent across the wire.

Using Oracle SQL/XML Generation Functions

The simplest way to benefit from moving XML processing to the database is to use Oracle SQL/XML functions, which allow you to build SQL queries generating XML from relational data.

Turning back to the preceding sample, you might, for example, rewrite the query issued against the database so that it retrieves the generated employees XML document that is ready to be transformed into HTML with the PHP XSL extension functions.
Diagrammatically, this might look like the following figure:

The explanation of the steps in the figure is the following:

- **Step 1**: The script issues the query containing SQL/XML functions so that it retrieves an XML document generated by the database server.
- **Step 2**: The database server generates the XML document, based on the query issued by the script in step 1.
- **Step 3**: The script transforms the XML document retrieved from the database into HTML format with the help of the PHP XSL extension functions.
- **Step 4**: The script posts the HTML page generated in step 3 to the user's browser.

In this scenario, you move some XML processing from the web server to the database server. In particular, the XML document is now generated on the database server with the help of the SQL/XML generation functions specified in the query, rather than generating that document on the web server with the PHP DOM extension functions as it was in the scenario depicted in the figure shown in the Transforming and Processing XML with XSLT section earlier in this chapter.

The following listing contains the `SQLXMLQuery.php` script that implements the above scenario. So, the script issues the query that makes Oracle generate the employees XML document, thus retrieving the employees XML document that is ready to be transformed with XSLT. The following script provides an example of using Oracle SQL/XML functions to generate XML from relational data. Using these functions lets you move the processing required to generate the employees XML document from the web server to the database server.

<?php
    //File: SQLXMLQuery.php
    if(!$rsConnection = oci_connect('hr', 'hr', '//localhost/orcl')) {
        $err = oci_error();
        trigger_error('Could not establish a connection: ' . $err['message'], E_USER_ERROR);
    }
    $dept_id = 90;
    $query = 'SELECT XMLELEMENT("EMPLOYEES",
        XMLAgg(
            XMLELEMENT("EMPLOYEE",
                XMLATTRIBUTES(employee_id AS "id"),
                XMLFOREST(last_name as "ENAME", salary as "SALARY")))
        AS result
        FROM employees WHERE department_id=:deptid';
    $stmt = oci_parse($rsConnection, $query);
    oci_bind_by_name($stmt, ':deptid', $dept_id);
    if (!oci_execute($stmt)) {
        $err = oci_error($stmt);
        trigger_error('Query failed: ' . $err['message'], E_USER_ERROR);
    }
    $xmlDoc = oci_fetch_assoc($stmt);
    $domxml = new DOMDocument();
    $domxml->loadXML($xmlDoc['RESULT']);
    $domxsl = new DOMDocument();
    $domxsl->load('employees.xsl');
    $proc = new XSLTProcessor;
    $xsl = $proc->importStylesheet($domxsl);
    $rslt = $proc->transformToXml($domxml);
    print $rslt;
?>

As you can see, the SQLXMLQuery.php script, unlike the DOM.php script discussed earlier in this chapter, does not use the PHP DOM functions to generate the employees XML document from scratch, based on the result set retrieved from the database. Instead, it issues a query that instructs the database server to generate that XML document. After executing the query, you fetch the result of the query and then load it to the newly created DOM document.

Next, you load the employees.xsl XSL stylesheet discussed in the Transforming and Processing XML with XSLT section earlier, assuming that this file resides in the same directory where you saved the SQLXMLQuery.php script discussed here.

Then, you create an XSLT processor, in which you import the employees.xsl stylesheet loaded into a DOM document. After performing the XSL transformation, you print the resultant HTML page.
When you run the `SQLXMLQuery.php` script, it should output a page that looks like the one shown in the figure in the *Transforming and Processing XML with XSLT* section.

**Moving All the XML Processing into the Database**

In the preceding example, the database server performs only a part of the XML processing while the rest is still performed by the PHP engine. Specifically, the database server generates an `employees` XML document based on the records from the `hr.employees` table, and the PHP script then transforms that document with XSLT into HTML format with the PHP XSL extension functions.

As an efficient alternative to PHP's XSLT processor, you might use Oracle's XSLT processor, thus benefiting from performing XSL transformations inside the database.

The following figure depicts the scenario where both generating XML and then transforming it into HTML take place inside the database.

There are several advantages to performing XSLT transformations, as well as many other XML processing operations, inside the database. These advantages are outlined at the beginning of the *Performing XML Processing inside the Database* section earlier in this chapter.

The explanation of the steps in the figure is as follows:

- **Step 1:** The script issues the query containing SQL/XML functions so that it retrieves an HTML document generated by the database server.
- **Step 2:** The database server generates the XML document, based on the instructions in the query issued by the script in step 1.
- **Step 3:** The database server transforms the XML document into HTML with the XSL stylesheet specified in the query issued in step 1.
- **Step 4:** The script posts the HTML page retrieved from the database to the user's browser.

However, before you implement this scenario, you have to decide where to store the XSL stylesheet to be used for the XSL transformation. Obviously, retrieving the stylesheet from the web server before performing the transformation on the database server would be a bad idea in this case, since it would increase network overhead. In contrast, storing the stylesheet in the database would be the best solution for this situation.

When choosing the storage option for XSL stylesheets, you should bear in mind that an XSL stylesheet is in fact an XML document. So, it would be a good idea to choose one of the XML storage options available in Oracle database.

### Storing XML Data in the Database

When using the database as a persistent storage for XML, you have several storage options. While all these options are discussed in the *Database Storage Options for XML Data in Oracle Database* section later in this chapter, this section provides a simple example of how you might store XML documents in an XMLType column in a database table as Character Large Object (CLOB) values. Once created, such a table can be used for storing different XML documents, including XSL stylesheets.

However, before creating this table you might want to create a new database schema. To create that schema and grant it all the required privileges, you might execute the SQL statements shown below:

```sql
CONN /as sysdba
CREATE USER xmlusr IDENTIFIED BY xmlusr;
GRANT connect, resource TO xmlusr;
```

Once the `xmlusr` schema is created and all the privileges required to work with it are granted, you can create the `XSLTstylesheets` table under this schema and populate it with the data. You might achieve this by issuing the SQL statements shown next:

CONN xmlusr/xmlusr

CREATE TABLE XSLTstylesheets {
    id NUMBER,
    stylesheet XMLType
};

INSERT INTO XSLTstylesheets VALUES (
    1,
    XMLType(
        '<?xml version="1.0" encoding="utf-8" ?>
        <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
        Transform">
            <xsl:template match="/">
                <html>
                    <head>
                        <title>Employees</title>
                    </head>
                    <body>
                        <font face="Arial">
                            <h2>List of employees from employees.xml</h2>
                            <table border="1" cellspacing="0" cellpadding="5">
                                <tr>
                                    <th><b>EMPLOYEE ID</b></th>
                                    <th><b>LAST NAME</b></th>
                                    <th><b>SALARY</b></th>
                                </tr>
                                <xsl:for-each select="EMPLOYEES">
                                    <xsl:for-each select="EMPLOYEE">
                                        <tr>
                                            <td><xsl:value-of select="@id"/></td>
                                            <td><xsl:value-of select="ENAME"/></td>
                                            <td><xsl:value-of select="SALARY"/></td>
                                        </tr>
                                    </xsl:for-each>
                                </xsl:for-each>
                            </table>
                        </font>
                    </body>
                </html>
            </xsl:template>
        </xsl:stylesheet>
    ));
COMMIT;

As you can see, inserting a new row into a table that contains an XMLType column is similar to inserting a new row into any other table—you use an `INSERT` statement and then issue the `COMMIT` to make the changes permanent. The only thing to notice here is that you have to explicitly convert the string representing an XML document to an XMLType value before inserting it to an XMLType column.

In this example, you insert only one row into the newly created `XSLTstylesheets` table. The `stylesheet` column of XMLType in this row includes the employees XSL stylesheet discussed in the *Transforming and Processing XML with XSLT* section earlier in this chapter. Once you have stored this stylesheet in the `XSLTstylesheets` table, you can access it with a `SELECT` statement when connected as `xmlusr/xmlusr`.

However, before you can move on to a script that will implement the scenario depicted in the figure shown in the *Moving All the XML Processing into the Database* section earlier in this chapter, you need to grant the `SELECT` privilege on the `hr.employees` table to the `xmlusr` database schema. This can be done by issuing the following statements from SQL*Plus:

```
CONN /as sysdba
GRANT SELECT ON hr.employees TO xmlusr;
```

By granting the `SELECT` privilege on the `hr.employees` table to `xmlusr` you permit the applications that will connect to the database through this schema to access data stored in the table.

### Performing XSLT Transformations inside the Database

Now that you have the employees XSL stylesheet stored in the database and the `xmlusr` schema is permitted to access the `hr.employees` table, you can create a script that will instruct the database to build an HTML page based on the data stored in `hr.employees`.

The following listing contains the source code for such a script.

```php
<?php
    //File: DBServerXSLTrans.php
    if (!$rsConnection = oci_connect('xmlusr', 'xmlusr', 'localhost/orcl')) {
        $err = oci_error();
        trigger_error('Could not establish a connection: '. $err['message'], E_USER_ERROR);
    }
    $dept_id = 90;
    $query = 'SELECT XMLtransform(x.xmlcol,';  
```
(SELECT stylesheet FROM XSLTstylesheets WHERE id = 1)).getStringVal()
AS result FROM
(SELECT XMLELEMENT("EMPLOYEES",
    XMLAgg(
        XMLELEMENT("EMPLOYEE",
            XMLATTRIBUTES(employee_id AS "id"),
            XMLFOREST(last_name AS "ENAME", salary AS "SALARY")
    )
) ) AS xmlcol
FROM hr.employees WHERE department_id=:deptid) x';
$stmt = oci_parse($rsConnection,$query);
oci_bind_by_name($stmt, ':deptid', $dept_id);
if (!oci_execute($stmt)) {
    $err = oci_error($stmt);
    trigger_error('Query failed: '.$err['message'], E_USER_ERROR);
}
$xmlDoc = oci_fetch_assoc($stmt);
$dom = new DOMDocument();
$dom->loadXML($xmlDoc['RESULT']);
echo $dom->saveXML();
?>

As you can see, the select list of the SELECT statement used in the
DBServerXSLTrans.php script includes the XMLtransform SQL/XML function.
This function is used here to apply the employees XSL stylesheet retrieved from
the XSLTstylesheets table by the subquery to the employees XML document
generated by the subquery defined in the FROM clause of the query. The result of this
transformation should be an HTML page, which you load into a new DOMDocument
object and then display it in the browser. When displayed, the generated HTML page
should look like the figure shown in the Transforming and Processing XML with XSLT
section shown earlier in this chapter.

Building PHP Applications on Oracle
XML DB

The preceding example shows how you might move the XML processing performed
by your PHP/Oracle application from PHP to Oracle, thus taking advantage of the
optimizations provided by the Oracle database server. In particular, you saw how to
generate an XML document from scratch and apply an XSL transformation inside the
database, rather than performing these operations with PHP.

In fact, Oracle XML Database provides much more functionality than what the sample demonstrates.

Oracle XML DB refers to the set of Oracle Database XML technologies integrated with the relational database server, providing high-performance XML storage, retrieval, and processing.

The most significant features of Oracle XML DB, which make Oracle database ideal for XML-enabled database-driven applications, are listed below:

- Ability to store, retrieve, update, and transform XML data through the SQL and PL/SQL interfaces.
- Ability to perform XML operations on SQL data without physically migrating it into XML format.
- Oracle XML DB repository lets you manipulate XML content stored in the database with the standard Internet protocols, such as FTP, HTTP, and WebDAV.
- Support for the Worldwide Web Consortium (W3C) XML Schema Recommendation: [http://www.w3.org/TR/xmlschema-0/](http://www.w3.org/TR/xmlschema-0/), allowing you to validate XML documents against appropriate XML schemas registered in the database.
- XML-specific optimizations, reducing the cost of performing XML processing inside the database.

The subsections that follow show how you can make use of these features when building XML-enabled PHP/Oracle applications.

### Using Oracle Database for Storing, Modifying, and Retrieving XML Data

With Oracle XML DB, you have various XML storage and XML processing options allowing you to achieve the required level of performance and scalability. One of the most interesting things about Oracle XML DB is that it allows you to perform SQL operations on XML data as well as XML operations on relational data, thus bridging the gap between the SQL and XML worlds.

Database Storage Options for XML Data in Oracle Database

When storing XML in Oracle database, you can choose between several storage options. The general XML storage options available in Oracle database are outlined in the following table:

<table>
<thead>
<tr>
<th>XML storage option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLType CLOB</td>
<td>Storing an XML document as a CLOB is a good idea if this document will be normally retrieved as a whole and updated by rewriting the entire document, rather than by performing piece-wise updates.</td>
</tr>
<tr>
<td>Native XMLType</td>
<td>With structured storage, an XML document must conform to a certain XML schema and is stored in the database as a set of SQL objects, which provides excellent DML performance in most situations.</td>
</tr>
<tr>
<td>(Structured storage)</td>
<td></td>
</tr>
<tr>
<td>XMLType views</td>
<td>Using XMLType views lets you operate on XML data created on top of relational data, thus allowing you to construct XML representations of that data.</td>
</tr>
</tbody>
</table>

The following figure will help you understand better the ideas behind the storage methods outlined in the table.
As you can see from the previous figure, when using CLOB storage for XMLType data, an XML document is stored in an XMLType column or table as a complete text document. Hence, updating an XML document stored as an XMLType CLOB is a very expensive operation that involves DOM parsing the document, performing the update operation on the DOM representation of the document, serializing the updated document back into text, and finally replacing it. Storing XML in CLOBs can be efficient when, for example, you're dealing with large XML documents, which are not updated frequently, and which you are going to retrieve as a whole.

In the preceding sample, you use XMLType CLOB storage for the employees XSL stylesheet, storing it in the stylesheet XMLType column of the XSLTstylesheets table, as discussed in the Storing XML Data in the Database section earlier. The XMLType CLOB storage is the best choice in that example because the only operation you are supposed to perform on the employees XSL stylesheet frequently is retrieving it as a whole when it comes to transforming an employees XML document into HTML.

In contrast, native XMLType storage, also known as structured or shredded storage, can be very efficient when you perform update operations on XML data frequently. This type of storage is created automatically by Oracle when registering an XML schema against the database.

Based on the information in an XML schema, Oracle creates a set of SQL object types and XMLType tables to be used for managing and storing the contents of XML documents conforming to that XML schema. Before storing, a document is broken up, and its contents are stored as an instance of the appropriate object type generated during the XML schema registration process.

This approach makes it possible for Oracle XML DB to rewrite SQL statements issued to access or manipulate XML schema-based XML data to purely relational SQL statements, thus allowing for efficient processing of XML data.

XMLType views can be useful when you need to wrap existing relational data in XML format without physically migrating it into XML. In fact, you can define an XMLType view not only on relational tables and views but also on XMLType ones. For examples on using XMLType views, see the Using XMLType Views section later in this chapter.

As you can see, all the XML storage options presented in the table are based on XMLType. The following section discusses this native Oracle datatype in detail.

Using XMLType for Handling XML Data in the Database

Being an object type, XMLType can not only be used to store XML data in the database but also to operate on that data via its built-in methods. Regardless of the storage model you choose, XMLType provides a set of XML-specific methods to operate on XMLType instances.

The most commonly used methods of XMLType are listed in the following table:

<table>
<thead>
<tr>
<th>XMLType method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>existsNode</td>
<td>Checks whether the XML document in a given XMLType instance contains a node that matches the XPath expression passed as the parameter. If the specified node is found, it returns 1; otherwise, it returns 0.</td>
</tr>
<tr>
<td>extract</td>
<td>Extracts a node or nodes from the XML document in an XMLType instance, based on the XPath expression passed as the parameter. Returns the result nodes as an XMLType instance.</td>
</tr>
<tr>
<td>createSchemaBasedXML</td>
<td>Explicitly associates an XML document in an XMLType instance with a registered XML schema specified in the parameter. You might want to perform this operation when inserting an XML document into an XML schema-based XMLType column or table.</td>
</tr>
<tr>
<td>schemaValidate</td>
<td>Validates an XML document in an XMLType instance against an XML schema specified in the parameter. On success, the status of the document is changed to VALIDATED; otherwise, an error is raised.</td>
</tr>
<tr>
<td>transform</td>
<td>Transforms an XML document in an XMLType instance with the XSL stylesheet specified in the parameter. Returns the resultant document as an XMLType instance.</td>
</tr>
</tbody>
</table>

You saw an example of using an XMLType method in the preceding sample application. In particular, in the DBServerXSLTrans.php script discussed in the Performing XSLT Transformations inside the Database section you use the getStringVal method of XMLType to retrieve the generated XHTML data as a VARCHAR value, so that it can then be loaded in a DOMDocument instance. If you recall, the query used in the DBServerXSLTrans.php script looks as follows:

```php
$query = 'SELECT XMLtransform(x.xmlcol, (SELECT stylesheet FROM XSLTstylesheets WHERE id = 1)).getStringVal() AS result FROM (SELECT XMLELEMENT("EMPLOYEES", XMLAgg((
```

XML-Enabled Applications

```sql
XMLELEMENT("EMPLOYEE",
  XMLATTRIBUTES(employee_id AS "id"),
  XMLFOREST(last_name AS "ENAME", salary AS "SALARY")
) AS xmlcol
FROM hr.employees WHERE department_id=:deptid) x';
```

To see another XMLType method in action, namely `transform`, you might rewrite the above SQL statement as follows:

```sql
$query = 'SELECT x.xmlcol.transform((SELECT stylesheet FROM XSLTstylesheets WHERE id = 1)).getStringVal()
AS result FROM
(SELECT XMLEDIMENT("EMPLOYEES",
XMLAgg(
  XMLELEMENT("EMPLOYEE",
    XMLATTRIBUTES(employee_id AS "id"),
    XMLFOREST(last_name AS "ENAME", salary AS "SALARY")
  )
) AS xmlcol
FROM hr.employees WHERE department_id=:deptid) x';
```

In the above query, you use the `transform` XMLType method as an alternative to the `XMLtransform` SQL function used in the original query. Since both `transform` and `XMLtransform` have the same functionality, the above queries will produce the same general result.

XMLType methods `extract` and `existsNode` can be used instead of the SQL functions having the same names.

As you can see, the above queries operate on relational data, and transform it into XML format with SQL/XML generation functions. Before looking at the approaches you can take to retrieve XML data stored in the database natively, however, it would be a good idea to look at how you can create an XMLType storage in Oracle XML DB. The following section discusses how you can do this with the help of the XML Schema feature.

Using XML Schemas

The simplest way to create an XMLType storage structure in Oracle XML DB is by registering an appropriate XML schema against the database. As a part of the registration process, Oracle automatically creates the storage for a particular set of XML documents, based on the information provided by the schema.

An XML schema can be thought of as the metadata describing a certain class of XML documents. So, an XML document conforming to a particular XML schema can be considered as an instance of this XML schema.

You might want to use an XML schema for:

- Building the storage for XML documents conforming the schema
- Setting up business rules on XML content of conforming documents
- Validating XML documents conforming to the schema

However, before you can use an XML schema, you have to create and then register it against the database. Both these tasks can be accomplished in one step with the `registerschema` procedure from the `DBMS_XMLSCHEMA` PL/SQL package. For example, to register an XML schema to which the following employee XML document conforms:

```xml
<EMPLOYEE id="100">
  <ENAME>King</ENAME>
  <SALARY>24000</SALARY>
</EMPLOYEE>
```

You might issue the following statements:

```
CONN /as sysdba
GRANT ALTER SESSION TO xmlusr;
CONN xmlusr/xmlusr
BEGIN
DBMS_XMLSCHEMA.registerschema(
  'employee.xsd',
  '<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:element name="EMPLOYEE" type="EMPLOYEE_TYP"
    xdb:defaultTable="EMPLOYEES"
    xdb:columnProps=""
  
As you can see, the `DBMS_XMLSCHEMA.registerschema` procedure takes two arguments. The first one is the string representing the name under which you want to register the schema against the database, and the other one is the document containing the schema itself.

In this example, the root element of the XML schema includes two namespace declarations, namely the XML schema namespace declaration and Oracle XML DB namespace declaration. To denote these namespaces, you use prefixes: `xs` and `xdb` respectively.

By including the XML schema namespace declaration, you obtain the ability to use the elements and attributes defined in this namespace, as well as the data types defined by the XML Schema language. For example, in the above example you specify the `positiveInteger` XML Schema language data type for the `id` attribute of the `EMPLOYEE` element.

The Oracle XML DB namespace lets you use annotations in the schema. For example, you use the `xdb:defaultTable` annotation to tell Oracle to use the specified table name when generating an XMLType table that will be used for storing XML documents conforming to the schema, rather than using a system-generated name for that table. In this particular example, you specify `EMPLOYEES` as the name for this table.

Another interesting annotation used in this XML schema is the xdb:columnProps. In this example, you use this annotation to define a primary key on the EMPLOYEE element's id attribute mapped to the EMPNO attribute of the EMPLOYEE_T SQL object type.

By including the xdb:SQLName annotation you make sure that the name of the generated SQL object type will be EMPLOYEE_T.

Finally, note the use of the flags passed to the DBMS_XMLSCHEMA.registerschema procedure:

```
TRUE,
TRUE,
FALSE,
TRUE
```

The above flags indicate the following (in the same order as they appear in the listing):

- The schema is generated as local (visible only to the database user who created it)
- Appropriate SQL object types are generated
- Java beans are not generated
- Default tables are generated

After registering the schema, you might want to look at the database object generated during the registration. The following listing contains the SQL statements that you might issue from SQL*Plus to make sure that Oracle generated the objects annotated in the schema. For convenience, the listing also contains the output produced.

```
DESC employee_t
employee_t is NOT FINAL

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_XDBPD$</td>
<td></td>
<td>XDB.XDB$RAW_LIST_T</td>
</tr>
<tr>
<td>EMPNO</td>
<td></td>
<td>NUMBER(38)</td>
</tr>
<tr>
<td>ENAME</td>
<td></td>
<td>VARCHAR2(4000)</td>
</tr>
<tr>
<td>CHAR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

DESC employees

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>---------------</td>
<td>-------</td>
<td>-----------------------------</td>
</tr>
</tbody>
</table>
```

TABLE of

SYS.XMLTYPE(
    XMLSchema "employee.xsd"
    Element "EMPLOYEE"
)
    STORAGE Object-relational TYPE "EMPLOYEE_T"

As you can see, Oracle generated the employee_t object type and employees
XMLType table based on this object type, as a part of the XML schema registration
process. Note that the names of the generated objects have been defined in the
schema. If you recall, you set the value of the xdb:SQLName attribute of global
element EMPLOYEE to EMPLOYEE_T, and the xdb:defaultTable attribute to
EMPLOYEES.

It's interesting to note that the names of database objects generated during
the XML schema registration process are case sensitive. However, since
SQL is case insensitive, you can refer to these objects in SQL disregarding
the case of their names. The names of XML elements and attributes
specified in an XML schema are also case sensitive. However, unlike SQL,
XML is case-sensitive, which means you must refer to XML elements and
attributes in XML code using the case with which they were defined in
the XML schema.

Now that you have defined the XMLType storage for employee XML documents,
you might want to load some data into the employees XML schema-based XMLType
table generated during the schema registration. The simplest way to do this is to use
the INSERT SQL statement, as follows:

```sql
CONN xmlusr/xmlusr
INSERT INTO employees VALUES(
    XMLType(''
        <EMPLOYEE id="100">
            <ENAME>King</ENAME>
            <SALARY>24000</SALARY>
        </EMPLOYEE>'
    ).createSchemaBasedXML('employee.xsd')
);
COMMIT;
```

In the above example, you use the createSchemaBasedXML method of XMLType
to explicitly identify the employee XML schema when inserting a new row into the
employees table.
Now, if you try to issue the `INSERT` statement shown in the listing again, you will receive the following error message:

```
ERROR at line 1:
ORA-00001: unique constraint (XMLUSR.EMP_PKEY) violated
```

As you can see, an attempt to insert the same row into the `employees` table fails due to a `EMP_PKEY` primary key constraint violation. If you recall, you define the `EMP_PKEY` primary key on the `id` attribute of the `EMPLOYEE` element in the XML schema registered as discussed in this section earlier. This constraint makes it impossible to insert two employee XML documents with the same ID into the `employees` table.

Another way to load data into the `employees` table is via one of the internet protocols supported by Oracle XML DB. This mechanism is discussed in the `Taking Advantage of Standard Internet Protocols` section later in this chapter.

Finally, it's worth noting that you can always delete a registered XML schema along with all the database objects generated during its registration. For example, to delete the `employee` XML schema registered as discussed earlier in this section, you might issue the following PL/SQL block:

```
CONN xmlusr/xmlusr

BEGIN
    DBMS_XMLSCHEMA.deleteSchema(
      SCHEMABURL => 'employee.xsd',
      DELETE_OPTION => dbms_xmlschema.DELETE_CASCADE_FORCE);
END;
/
```

Since the `employees` table is used in the subsequent examples, make sure to register the `employee` XML schema again as discussed earlier in this section. Also make sure to insert a row into the table as shown earlier in this section.

Besides deleting the `employee` XML schema, the above code deletes the `employee_t` object type and `employees` XMLType table generated during the schema registration process.

Retrieving XML Data

To retrieve XML data from an XMLType table, you can use a `SELECT` SQL statement, just as you would if you had to query a relational table. For example, to select the employee with the id set to 100 from the `employees` XMLType table discussed in the preceding section, you might issue the following query from SQL*Plus when connected as `xmlusr/xmlusr`:

```sql
SELECT * FROM employees x WHERE existsNode(value(x), '/EMPLOYEE/@id="100"') = 1;
```

This query should produce the following output:

```
SYS_NC_ROWINFO$
------------------------
<EMPLOYEE id="100">
    <ENAME>King</ENAME>
    <SALARY>24000</SALARY>
</EMPLOYEE>
```

The `QueryXML.php` script defined below shows how the above query might be issued from PHP.

```php
<?php
    // File: QueryXML.php
    if(!$rsConnection = oci_connect('xmlusr', 'xmlusr', '/localhost/orcl')) {
        $err = oci_error();
        trigger_error('Could not establish a connection: '.$err['message'], E_USER_ERROR);
    }
    $xpath_exp = '/EMPLOYEE/@id="100"';
    $query = 'SELECT value(x).GetStringVal() as RESULT
              FROM employees x
              WHERE existsNode(value(x), :xpath) = 1';
    $stmt = oci_parse($rsConnection,$query);
    oci_bind_by_name($stmt, ':xpath', $xpath_exp);
    if (!oci_execute($stmt)) {
```
In the above script, you set the $xpath_exp variable to the XPath expression that points to the EMPLOYEE node whose id attribute is set to 100. This variable is then bound to the :xpath placeholder.

Note the use of the value(x) pseudocolumn in the select list of the query. In this example, value(x) is used to access the XMLType object representing an employee XML document retrieved by the query. You use the getStringVal XMLType method to convert the retrieved XML document into a string, so that it can be loaded into a DOMDocument.

When you run the QueryXML.php script shown in the listing, it should produce the following output:

```xml
<?xml version="1.0" ?>
<EMPLOYEE id="100">
  <ENAME>King</ENAME>
  <SALARY>24000</SALARY>
</EMPLOYEE>
```

If your browser omits XML tags, though, you will see the following:

```
King   2400
```

While the existsNode SQL function used in the preceding example checks for the existence of elements based on the XPath expression, the extractValue SQL function lets you extract the value of a node or attribute conforming to the specified XPath expression. So, the extractValue SQL function lets you access XML data, receiving results similar to those received when querying relational data.
The following query is a simple example of `extractValue` in action:

```sql
SELECT extractValue(OBJECT_VALUE, '/EMPLOYEE/ENAME')
ENAME FROM employees WHERE existsNode(OBJECT_VALUE,
  '/EMPLOYEE/@id="100"') = 1;
```

As you can see, the query extracts the value of the `ENAME` node under the `EMPLOYEE` node whose `id` attribute is set to `100`. Note the use of the `OBJECT_VALUE` pseudocolumn in the query. This pseudocolumn is an Oracle Database 10g alternative to `value(x)`. In this query, you use `OBJECT_VALUE` to access an `employee` `XMLType` object retrieved from the `employees` table.

When issued from SQL*Plus, the above query should return the following result:

```
ENAME
--------------------------------------------
King
```

You might rewrite the query to use the `extract` and `existsNode` `XMLType` methods as follows:

```sql
SELECT x.OBJECT_VALUE.extract('/EMPLOYEE/ENAME/text()').getStringVal()
ENAME FROM employees x
WHERE x.OBJECT_VALUE.existsNode('/EMPLOYEE/@id="100"') = 1;
```
To test this query with PHP, you might write the `extractXML.php` script shown below:

```php
<?php
// File: extractXML.php
if (!$rsConnection = oci_connect('xmlusr', 'xmlusr', '//localhost/orcl')) {
    $err = oci_error();
    trigger_error('Could not establish a connection: '. $err['message'], E_USER_ERROR);
}
$id = 100;
$exist_exp = '/EMPLOYEE/@id='.$id.';
$extr_exp = '/EMPLOYEE/ENAME/text()';
$query = 'SELECT x.OBJECT_VALUE.extract(:extr).getStringVal() ENAME
         FROM employees x
         WHERE x.OBJECT_VALUE.existsNode(:exist)=1';
$stmt = oci_parse($rsConnection,$query);
oci_bind_by_name($stmt, ':extr', $extr_exp);
oci_bind_by_name($stmt, ':exist', $exist_exp);
if (!oci_execute($stmt)) {
    $err = oci_error($stmt);
    trigger_error('Query failed: '. $err['message'], E_USER_ERROR);
}
$xmlDoc = oci_fetch_assoc($stmt);
print '<h2>The name of employee whose id='.$id.' is:</h2>'; print $xmlDoc['ENAME'];
?>
```

The query used in the script represents a simple example of using the `extractValue` SQL function. Usually, `extractValue` is used in complex SQL statements in which the data extracted from XML is then used in `INSERT` or `UPDATE` operations performed on relational tables.

### Accessing Relational Data Through `XMLType` Views

Using relational tables to store shredded XML documents allows you to take advantage of both the Oracle XML technologies and Oracle database relational technologies when developing XML-enabled applications.

For example, you can easily implement fine-grained access when working with XML content built upon relational data. In Chapter 9 Web Services, you will see an example of how to secure XML data, based on the row-level security implemented on the relational data upon which that XML data is built.

In the preceding sections, you saw several examples of how to construct XML from SQL data with the help of SQL/XML generation functions. In the following sections, you will learn how to simplify the development of XML-enabled PHP/Oracle applications with XMLType views built upon relational tables.

### Using XMLType Views

XMLType views provide a convenient way to construct XML representations of relational data without physically migrating that data into XML. Once written, an XMLType view may be used in various queries, making them simpler and so increasing their readability.

Turning back to the `SELECT` statement used in the `SQLXMLQuery.php` script discussed in the Using Oracle SQL/XML Generation Functions section earlier in this chapter, you might create an XMLType view based on that statement as shown below.

```sql
CONN /as sysdba
GRANT CREATE ANY VIEW TO xmlusr;
CONN xmlusr/xmlusr;
CREATE VIEW EmpsXML AS
    SELECT XMLELEMENT("EMPLOYEES",
        XMLAgg(
            XMLELEMENT("EMPLOYEE",
                XMLATTRIBUTES(employee_id AS "id"),
                XMLFOREST(last_name AS "ENAME", salary AS "SALARY")
            )
        ) AS xmlcol,
        department_id AS dept_id
    FROM hr.employees GROUP BY department_id;
```

In this example, you start by granting the `CREATE VIEW` privilege to the `xmlusr` database schema and then, when connected as `xmlusr/xmlusr`, create the `EmpsXML` view based on the query that uses SQL/XML functions to generate XML from the data stored in the `hr.employees` relational table.
The good thing about the EmpsXML view is that it hides the details of generating an employees XML document, thus letting you write simpler and more readable queries. With it, the query used in the SQLXMLQuery.php script might be rewritten as follows:

```php
$query = 'SELECT xmlcol as RESULT FROM EmpsXML WHERE dept_id=:deptid';
```

Before running the updated SQLXMLQuery.php script, make sure to specify the xmlusr/xmlusr schema in the oci_connect function at the beginning of the script as follows:

```php
$rsConnection = oci_connect('xmlusr', 'xmlusr', '//localhost/orcl'))
```

Also, you might rewrite the query string used in the DBServerXSLTrans.php script discussed in the Performing XSLT Transformations inside the Database section earlier in this chapter as follows:

```php
$query = 'SELECT XMLtransform(x.xmlcol, (SELECT stylesheet FROM XSLTstylesheets WHERE id = 1)).getStringVal() AS result FROM (SELECT * FROM EmpsXML WHERE dept_id=:deptid) x';
```

As you can see, the above query is three times smaller than the one originally used in the DBServerXSLTrans.php script.

**Creating XML Schema-Based XMLType Views**

While the Using XML Schemas section earlier in this chapter focuses on how the XML Schema feature of Oracle XML DB can be used to create an XML schema-based storage structure, this section discusses how XML schema functionality might be used when working with existing relational data, without having to change the physical structure of that data.

Creating an XML schema-based XMLType view is the most common way to take advantage of XML schema functionality when dealing with data stored relationally.

However, before you create an XML schema-based XMLType view, you must have the appropriate XML schema created and registered against the database. By executing the statement shown overleaf, you create and register the emp.xsd XML schema on which you will then create an XMLType view.

XML-Enabled Applications

CONN xmlusr/xmlusr
BEGIN
DBMS_XMLSCHEMA.registerschema(
  'emp.xsd',
  '<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xdb="http://xmlns.oracle.com/xdb">
    <xs:element name="EMPLOYEE" type="EMP_TYP"/>
    <xs:complexType name="EMP_TYP" xdb:SQLType="EMP_T"
      xdb:maintainDOM="false">
      <xs:sequence>
        <xs:element name="ENAME" type="enameType" xdb:SQLName="ENAME"
          xdb:SQLType="VARCHAR2"/>
        <xs:element name="SALARY" type="salaryType" xdb:SQLName="SALARY"
          xdb:SQLType="NUMBER"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:positiveInteger"
        xdb:SQLName="EMPNO"
        xdb:SQLType="NUMBER"/>
    </xs:complexType>
    <xs:simpleType name="salaryType">
      <xs:restriction base="xs:double">
        <xs:maxExclusive value="100000"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="enameType">
      <xs:restriction base="xs:string">
        <xs:minLength value="2"/>
        <xs:maxLength value="30"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>'
, TRUE, TRUE, FALSE, FALSE);
END;
/

As you can see from the listing, the EMPLOYEE element, which is the root element of the employee XML document described by this schema, is mapped to the EMP_T SQL object type. This object type will be automatically generated during schema registration as long as you set the DBMS_XMLSCHEMA.registerschema's fourth parameter, which is actually called GENTYPES, to TRUE.

At the same time, you set the sixth (GENTABLES) parameter to FALSE, thus instructing Oracle not to create any tables during schema registration. This makes sense in this case because you are going to map between this XML schema and an existing relational table later, with the help of an XMLType view.

After the PL/SQL block shown in the listing has been successfully executed, you might issue the DESC SQL command in order to make sure that the EMP_T object type was generated:

```
DESC EMP_T
```

This should return the following result:

```
EMP_T is NOT FINAL
Name                Null?    Type
------------------- -------- ----------------
EMPNO               NUMBER(38)
ENAME               VARCHAR2(4000
CHAR)
SALARY              NUMBER
```

Since DOM fidelity is not required when it comes to wrapping relational data in XML, you set the attribute maintainDOM to FALSE. As a result, the EMP_T type, unlike the EMPLOYEE_T type created as discussed in the Using XML Schemas section earlier, doesn't contain the SYS_XDBPDS attribute.

The XML schema defined in the listing contains an example of how to add a constraint to an element described in the schema, restricting its content to values matching a set of conditions. In particular, you restrict the value of node SALARY in all employee XML documents conforming to the schema to be less than 100 000. To achieve this, you use a maxExclusive element under the restriction element defined in turn under the simpleType element for the SALARY element.

The following listing shows how to set up an XML schema-based XMLType view based on the hr.employees relational table. The view created here conforms to the employee XML schema created as discussed at the beginning of this section.

```
CONN xmlusr/xmlusr
CREATE TABLE emps
AS SELECT employee_id, last_name, salary FROM hr.employees;
ALTER TABLE emps
ADD constraint EMP_PRIMARYKEY
PRIMARY KEY (employee_id);
CREATE OR REPLACE VIEW empSch_v OF XMLType
XMLSCHEMA "emp.xsd" ELEMENT "EMPLOYEE"
```

WITH OBJECT ID (extract(OBJECT_VALUE, '/EMPLOYEE/@id/text()').getNumberVal()) AS
SELECT EMP_T(e.employee_id, e.last_name, e.salary)
FROM emps e;

In the above listing, you start by creating relational table emps based on the hr.employees table. For simplicity, you include only three columns in the newly created table, while loading all the rows from hr.employees.

By specifying employee.xsd in the XMLSCHEMA clause and EMPLOYEE in the ELEMENT clause of the CREATE VIEW statement, you constrain a resultant row object in the view to be an instance of the element EMPLOYEE defined in the emp.xsd XML schema.

Since row objects in the empSch_v XMLType object view are synthesized from relational data, you must explicitly choose a set of unique identifiers to be used as object identifiers. In this example, in the WITH clause you specify the id attribute of the EMPLOYEE element as the object identifier because it is unique within the view row objects.

In the select list of the view, you explicitly convert the data retrieved from the relational table emps to the EMP_T SQL object type specified for the EMPLOYEE element in the emp.xsd XML schema.

Performing DML Operations on XML Schema-Based XMLType Views

Analyzing the underlying query of the empSch_v view discussed in the preceding section, you may note that each attribute of the EMP_T object used in the select list maps to a certain column of a single table, namely emps. What this means in practice is that the empSch_v view can be inherently updated, so you can perform DML operations against it without having to write INSTEAD-OF triggers.

The following figure gives a conceptual depiction of what occurs upon insertion of an XML document into an inherently updatable XML schema-based XMLType view.

Here is the explanation of the steps outlined in the previous figure:

- Step 1: PHP script posts a schema-based XML document to be inserted into an XML schema-based XMLType view.
- Step 2: Oracle checks whether the XML document being inserted into the view conforms to the XML schema on which the view is defined.
- Step 3: If the document conforms to the schema, it is shredded into relational data conforming to the underlying relational table.
- Step 4: The shredded XML document is inserted into the underlying relational table, as a new row.

Turning back to the `empSch_v` view, you might issue the following `INSERT` statement against it from SQL*Plus in order to make sure that the view actually allows you to perform `INSERT` operations on it:

```sql
INSERT INTO empSch_v VALUES(XMLType(
  '<EMPLOYEE
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="emp.xsd"
id="300">
...<XMLDOC
  ...
</XMLDOC>
</xs:schema>
</<?xml version="1?>
</<?xml version="1?>

Issuing a statement from SQL*Plus is always a good idea when you need to perform a quick test. All INSERT operations discussed in this section might be issued from within PHP code as well.

Note the use of the xsi:noNamespaceSchemaLocation attribute of the root document element EMPLOYEE in the above statement. This attribute is used to indicate the schema location. Alternatively, you might use the createSchemaBasedXML method of XMLType, as you did in the Using XML Schemas section when inserting a row into the employees table. However, in this example you would specify emp.xsd as the parameter of createSchemaBasedXML.

The data inserted through the empSch_v view can then be accessed not only through that view as XML, but also through its underlying table emps as relational data. For example, to retrieve the employee XML document inserted into the empSch_v view by the preceding query, you might use the following query:

```sql
SELECT * FROM empSch_v WHERE existsNode(OBJECT_VALUE,
   '/EMPLOYEE/@id="300"')=1;
```

On the other hand, to see a relational representation of the inserted document, you might issue the following query against the emps underlying table:

```sql
SELECT * FROM emps WHERE employee_id=300;
```

This should produce the following output:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Silver</td>
<td>12000</td>
</tr>
</tbody>
</table>

Now, what happens if you try to insert an employee XML document into empSch_v, which doesn't conform the emp.xsd XML schema? Say, for example, the value of the SALARY element in the inserted document exceeds the maximum allowable value specified for this element in the schema. For example, you might issue the following statement and see what happens:

```sql
INSERT INTO empSch_v VALUES (XMLType ("<EMPLOYEE
   '<ENAME> Silver</ENAME>
   '<SALARY>12000</SALARY>
   </EMPLOYEE>' ));
```

You might be surprised to see that the above statement works without any problem. This is despite the fact that the value of the `SALARY` element is restricted to be less than 100000, as you might recall from the listing in the *Creating XML Schema-Based XMLType Views* section, describing `emp.xsd` XML schema registration.

The fact is that Oracle performs only a partial validation when it comes to inserting an XML document into an XML schema-based XMLType table or column or view. In particular, it checks to see whether the structure of the XML document being inserted conforms to the appropriate XML schema and does not check the contents of the document.

So, to ensure that the `employee` XML documents inserted into the `empSch_v` view are fully compliant with the `emp.xsd` XML schema, you need to explicitly invoke an XML schema validation when performing `INSERT` operations. The simplest way to do this is to use a PL/SQL function that might be created as follows:

```sql
CONN xmlusr/xmlusr
CREATE OR REPLACE FUNCTION val_xml (xmldoc XMLType) RETURN XMLType IS tmpxml XMLType;
BEGIN tmpxml := xmldoc;
XMLTYPE.schemavalidate(tmpxml);
RETURN xmldoc;
END;
/```

Issuing a statement from SQL*Plus is always a good idea when you need to perform a quick test. All `INSERT` operations discussed in this section might be issued from within PHP code as well.

XML-Enabled Applications

After you have created the `val_xml` function, you might use it in `INSERT` operations issued against XML schema-based tables and views as follows:

```sql
INSERT INTO empSch_v VALUES(VAL_XML(XMLType('"
<EMPLOYEE
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="emp.xsd"
id="302">
<ENAME>Johnson</ENAME>
<SALARY>100000</SALARY>
</EMPLOYEE>
'))
);
```

Now, a full XML schema validation takes place. Since the value of the `SALARY` element in the above `employee` XML document is greater than the maximum allowable value defined in the schema, you should receive the following error message:

```
ERROR at line 1:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00292: value "100000" is greater than maximum "100000" (exclusive)
ORA-06512: at "SYS.XMLTYPE", line 345
ORA-06512: at "XMLUSR.VAL_XML", line 7
```

However, it is important to note that while a full XML schema validation allows you to validate both the structure and contents of an instance document, it comes at the cost of processing time and memory usage, thus adding overhead to your application and decreasing performance.

Using Oracle XML DB Repository

Another variation on accessing and manipulating XML content stored in Oracle database is provided by Oracle XML DB repository, which is an essential component of Oracle XML DB.

Oracle XML DB repository, also known as XML repository, is a hierarchically organized repository seamlessly integrated with Oracle Database, containing resources that can be manipulated using a file/folder/URL metaphor.

The most significant thing about XML repository is that it makes it possible to access and manipulate XML data in a number of different ways, including SQL, PL/SQL, and standard internet protocols, such as HTTP, FTP, and WebDAV. Graphically, it looks as shown in the following figure.

![XML Repository Diagram](image)

You may find it convenient to think of Oracle XML DB repository as a file system whose metadata and data are stored in the database. Like a conventional file system, Oracle XML DB repository contains resources: files and folders. However, in the case of XML repository, each resource also can be accessed through SQL.

Although XML repository is optimized for working with XML data, you can use it to store non-XML data as well. For example, you might store a collection of pictures there.

**Manipulating Repository Resources with PL/SQL**

Oracle XML DB provides PL/SQL package DBMS_XDB to access Oracle XML DB repository programmatically from within PL/SQL code.

---

XML-Enabled Applications

For example, to create a repository folder and then a resource in that folder, you might use the `DBMS_XDB.createFolder` and `DBMS_XDB.createResource` function respectively, as follows:

```plsql
CONN xmlusr/xmlusr;
DECLARE
  rslt BOOLEAN;
  xmldoc VARCHAR2(250) :=
    '<EMPLOYEE
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="employee.xsd"
      id="303">
      <ENAME>Locke</ENAME>
      <SALARY>7000</SALARY>
    </EMPLOYEE>'
BEGIN
  IF (NOT DBMS_XDB.existsResource('/public/xmlusr')) THEN
    rslt:=DBMS_XDB.createFolder('/public/xmlusr');
  END IF;
  IF (NOT DBMS_XDB.existsResource('/public/xmlusr/emps')) THEN
    rslt:=DBMS_XDB.createFolder('/public/xmlusr/emps');
  END IF;
  rslt := DBMS_XDB.createResource('/public/xmlusr/emps/emp303.xml',
    xmldoc);
  COMMIT;
END;
/
```

As you can see, when creating a resource, regardless of whether it is a file or folder, you must specify an absolute path to that resource. This is required because, as in a conventional file system, each resource in the XML repository is identified by a path and name.

Accessing Repository Resources with SQL

In fact, Oracle XML DB repository resources are stored in a set of database tables and indexes, which can be accessed via SQL. You are not supposed to access those tables directly. Instead, Oracle XML DB provides two public views `RESOURCE_VIEW` and `PATH_VIEW` through which you can access repository resources.

For example, you might issue the following query against the `RESOURCE_VIEW` view to access the employee XML document stored in the XML repository as `/public/xmlusr/emps/emp303.xml`, assuming that you have executed the PL/SQL block shown in the preceding section.

SELECT extract(r.RES, '/Resource/Contents/*').getStringVal()
RESULT FROM RESOURCE_VIEW r
WHERE equals_path(res, '/public/xmlusr/emps/emp303.xml') = 1;

This should produce the following result:

<EMPLOYEE
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="employee.xsd"
  id="303">
  <ENAME>Locke</ENAME>
  <SALARY>7000</SALARY>
</EMPLOYEE>

However, in this particular example you don't have to query RESOURCE_VIEW to retrieve the above XML document through SQL. Instead, you might issue the following query against the employees XMLType table:

SELECT extract(OBJECT_VALUE,'/').getStringVal()
RESULT FROM employees
WHERE existsNode(OBJECT_VALUE, '/EMPLOYEE/@id="303"') = 1;

You might be asking yourself: How could that have happened—a document uploaded into the XML repository appeared in an XMLType table? As you might recall from the listing describing the employee.xsd XML schema registration in the Using XML Schemas section, the employees XMLType table is specified as a default table in the employee.xsd XML schema and so it must have been generated during the schema registration process. Since the employee XML document inserted into the XML repository by the PL/SQL code as discussed in the preceding section is based on the employee.xsd XML schema, this document has been automatically inserted into the employees XMLType table.

Taking Advantage of Standard Internet Protocols

As mentioned, Oracle XML DB provides native support for standard internet protocols, such as HTTP(S), WebDAV, and FTP. Continuing with the preceding sample, you might, for example, upload another employee XML document into the XML repository with one of the above protocols, say, FTP.

Starting with Oracle Database 10g Release 2, FTP is disabled by default, for security reasons. This is achieved by setting the FTP port to 0. To enable FTP, you must manually set the FTP port number to an appropriate value, such as 2100, which is the default value in Oracle Database releases before 10g Release 2. Changing the FTP port to an appropriate value can be easily done with the help of Oracle Enterprise Manager, a graphical tool supplied with Oracle Database. For more information on Oracle Enterprise Manager, refer to Oracle documentation: chapter Getting Started with Oracle Enterprise Manager in the Oracle Database 2 Day DBA manual.

The `uploadXML.php` script shown below provides a simple example of how you might upload an XML document into the XML repository through FTP protocol.

```php
<?php
// File: uploadXML.php
$host='localhost';
$port=2100;
$timeout=30;
$db_user='xmlusr';
$db_pswd='xmlusr';
[root_dir='/public/xmlusr/emps';
$empid=304;
$file='emp'.$empid.'.xml';
$cnt=
'<?xml version="1.0"?>
<EMPLOYEE
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="employee.xsd"
 id="'.$empid.'">
 <ENAME>Polonski</ENAME>
 <SALARY>8200</SALARY>
 </EMPLOYEE>
';
$temp = tmpfile();
fwrite($temp, $cnt);
fflush($temp);
$ftpcon = ftp_connect($host, $port, $timeout);
$login = ftp_login($ftpcon, $db_user, $db_pswd);
ftp_chdir($ftpcon, $root_dir);
ftp_fput($ftpcon, $file, $temp, FTP_ASCII);
ftp_close($ftpcon);
?>
```

In the above script, you start by setting up the parameters required to connect to the FTP server running on the database server.
By including the value of the employee's `id` attribute in the name of the XML file to be uploaded into the XML repository, you ensure the uniqueness of file names within the repository folder that contains employee XML documents in single files.

Then, you create a temp file and write XML content in it. Then, you connect to the Oracle FTP server and upload the file into the XML repository.

After the `uploadXML.php` script is executed, to make sure that the employee XML document has been successfully uploaded into the XML repository, you might issue the following query from SQL*Plus:

```sql
CONN xmlusr/xmlusr
SELECT extract(OBJECT_VALUE,'/').getStringVal()
RESULT FROM employees
WHERE existsNode(OBJECT_VALUE, '/EMPLOYEE/@id="304"') = 1;
```

If everything is OK, this query should output the employee XML document uploaded by the `uploadXML.php` script:

```xml
<?xml version="1.0"?>
<EMPLOYEE
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="employee.xsd"
 id="304">
 <ENAME>Polonski</ENAME>
 <SALARY>8200</SALARY>
</EMPLOYEE>
```

So, after the `uploadXML.php` script is executed, the XML document inserted by the script becomes permanent immediately.

**Handling Transactions**

As you can see from the preceding example, changes made through internet protocols to repository resources become permanent once a single operation on a resource has been completed.

In contrast, all SQL and PL/SQL operations that you perform on the XML data stored in Oracle XML DB are transactional, irrespective of whether you're using XMLType storage or the XML repository. This allows you to combine SQL and PL/SQL statements operating on XML data within a logical unit of work and explicitly commit the transaction or roll it back if necessary.

Suppose you want to create a folder in the XML repository and then upload an XML document in it. The following code fragment shows how this can be done in PL/SQL:

---

The above PL/SQL code is transactional. If an error occurs during the execution of the createFolder function, then the results of execution of createFolder are discarded automatically.

On the other hand, when performing the same operations through FTP protocol, failure to create the resource doesn’t affect the results of preceding operation. So, the following fragment of PHP code is not transactional:

```
$login = ftp_login($ftpcon, $db_user, $db_pswd);
ftp_chdir($ftpcon, '/public/xmlusr/emps/');
ftp_mkdir($ftpcon, '303');
ftp_chdir($ftpcon, '/public/xmlusr/emps/303/');
ftp_fput($ftpcon, 'emp303.xml', $xmldoc, FTP_ASCII);
ftp_close($ftpcon);
```

Chapter 4 Transactions, presented earlier in this book, explains in detail how to use transactions in PHP/Oracle applications.

**Querying Data with Oracle XQuery**

Starting with Oracle Database 10g Release 2, you can take advantage of a full-featured native XQuery engine integrated with the database. With Oracle XQuery, you can accomplish various tasks involved in developing PHP/Oracle XML applications, operating on any kind of data that can be expressed in XML.
The following figure shows the data sources with which Oracle XQuery can work.

As you can see from the figure, XQuery can be used to query any data stored in the database and out of it.

Please note that XQuery is not available in the Express Edition of Oracle Database. For more information on Oracle XQuery, you can refer to Oracle documentation: chapter Using XQuery with Oracle XML DB in the Oracle XML DB Developer's Guide. You can also refer to the XQuery 1.0: An XML Query Language W3C Recommendation at http://www.w3.org/TR/xquery/.

Using XQuery to Construct XML from Relational Data

In the preceding sections, you saw several examples of how to construct XML representations of relational data using the SQL/XML generation function, as well as using object types when creating XMLType views on relational tables. In this section, you will learn how to build XML on relational data with XQuery.

Turning back to the Using Oracle SQL/XML Generation Functions section, you might modify the listing containing the SQLXMLQuery.php script that uses PHP DOM extension functions to produce an XML representation of relational data to use XQuery instead of those functions, as shown overleaf:

The Oracle SQL function `XMLQuery` can be used to construct or query XML data, based on an XQuery expression passed as the parameter. In this example, you use the `XMLQuery` function to generate an XML representation of some data stored in the relational table `hr.employees`.

You start the FLWOR expression used in this example with the `for` clause performing only one iteration.

```
<?php
    //File: XQuery.php
    if (!$rsConnection = oci_connect('hr', 'hr', '//localhost/orcl')) {
        $err = oci_error();
        trigger_error('Could not establish a connection: ' . $err['message'], E_USER_ERROR);
    }
    $dept_id=90;
    $query =
    'SELECT XMLQuery('.""'.for $j in 1
    return (</EMPLOYEE>
    <ENAME>{xs:string($i/LAST_NAME)}</ENAME>
    <SALARY>{xs:integer($i/SALARY)}</SALARY>
    </EMPLOYEE>)'."".
    'PASSING XMLElement("deptid", :deptid) AS "deptid"
    RETURNING CONTENT).getStringVal() RESULT FROM DUAL';
    $stmt = oci_parse($rsConnection,$query);
    oci_bind_by_name($stmt, ':deptid', $dept_id);
    if (!oci_execute($stmt)) {
        $err = oci_error($stmt);
        trigger_error('Query failed: ' . $err['message'], E_USER_ERROR);
    }
    $xmlDoc = oci_fetch_assoc($stmt);
    $domxml = new DOMDocument();
    $domxml->loadXML($xmlDoc['RESULT']);
    print $domxml->saveXML();
?>
```

FLWOR stands for `for`, `let`, `where`, `order by`, `return`—the clauses used when composing an XQuery expression.

Next, in the nested FLWOR expression that starts with the for clause, you iterate over the hr.employees rows selected based on the condition specified in the where clause. With the help of the ora:view XQuery function, you query relational table hr.employees, as it were an XMLType table, creating XML documents on the fly.

In the return clause of the FLWOR expression, you construct the EMPLOYEE nodes of the resultant EMPLOYEES document.

You bind dynamic variable deptid to an XQuery expression using the PASSING clause. This variable is used in the where clause of the nested FLWOR expression, restricting the retrieved employee records to those that belong to the specified department.

When executed, the XQuery.php script shown in the listing opposite should produce the same XML document as the one shown in the Creating XML with the DOM PHP Extension section at the beginning of this chapter. If you want to print the resultant XML document in HTML format, you might perform an XSL transformation before outputting it. To achieve this, you might replace the last line of code in the script:

```php
print $domxml->saveXML();
```

with the following lines:

```php
$domxsl = new DOMDocument();
$domxsl->load('employees.xsl');
$proc = new XSLTProcessor;
$xsl = $proc->importStylesheet($domxsl);
$rslt = $proc->transformToXml($domxml);
print $rslt;
```

Assuming that you have created the employees.xsl stylesheet as discussed in the Transforming and Processing XML with XSLT section earlier in this chapter, with the above replacement, the XQuery.php script shown in the listing should produce the HTML table as shown in the Transforming and Processing XML with XSLT section earlier.

**Breaking up XML into Relational Data**

While the preceding example shows how to construct an XML representation over relational data, the example in this section illustrates how you can shred XML data back into relational data. This reverse operation can be useful if your application works with relational data rather than XML, but the data which you work with, is stored in XML format.

Turning back to the `employees` XMLType table generated during the registration of the `employee.xsd` XML schema discussed in the Using XML Schemas section, you might use the SQL function `XMLTable` to shred the `employee` XML documents into individual columns of a virtual table, as shown below:

```sql
CONN xmlusr/xmlusr
SELECT emps.empno, emps.ename, emps.salary FROM employees,
    XMLTable('for $i in /EMPLOYEE
        return $i'
    PASSING OBJECT_VALUE
    COLUMNS empno NUMBER       PATH '/EMPLOYEE/@id',
        ename VARCHAR2(30) PATH '/EMPLOYEE/ENAME',
        salary NUMBER      PATH '/EMPLOYEE/SALARY') emps;
```

Assuming that you have followed the instructions in the preceding sections, the output generated by the query shown in the listing might look like this:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>101</td>
<td>Horka</td>
<td>25000</td>
</tr>
<tr>
<td>303</td>
<td>Locke</td>
<td>7000</td>
</tr>
<tr>
<td>304</td>
<td>Polonski</td>
<td>8200</td>
</tr>
</tbody>
</table>

In a real-world situation, in order to hide data complexity, you might find it useful to build a view on the query shown in the listing, so that the people designing SQL queries against that view have no idea they are dealing with data actually stored in XML format.

## Summary

When building XML-enabled PHP/Oracle applications, the database can not only be used as an efficient means for storing XML data, but also to operate on any kind of data that can be expressed in XML.

In this chapter you learned how to use XML techniques and technologies available in PHP and Oracle when building XML-enabled PHP/Oracle applications. In particular, you saw how to use PHP's XML extensions and how to take advantage of Oracle XML DB, a set of Oracle XML technologies making Oracle Database an ideal choice for data-driven XML applications.

By now you should have a good understanding of how you can use XML techniques and technologies available in PHP and Oracle to build robust XML-enabled PHP applications on Oracle Database. Armed with this knowledge, you will be able to understand better XML Web Services discussed in the next chapter.
Where to buy this book

You can buy PHP Oracle Web Development from the Packt Publishing website:

Free shipping to the US, UK, Europe, Australia, New Zealand and India.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.