Building XQuery-powered applications with PHP and Zorba

Add XQuery support to PHP with Zorba

Skill Level: Intermediate

Vikram Vaswani
Founder
Melonfire

03 Nov 2009

Updated 13 Oct 2010

Zorba is an open-source, robust, and standards-compliant XQuery processor. The Zorba extension in PHP provides an API to Zorba functions from within PHP, and thereby allows developers to add sophisticated XQuery processing to their PHP/XML applications. Examine the Zorba PHP API in detail, and how to use it for a variety of purposes.

11 Oct 2010 - *Per reader feedback, author revised the instructions in Installing Zorba. Under "Get products and technologies" in Resources, added a link to the SWIG toolkit.*

Introduction

Frequently used acronyms

- API: Application program interface
- DOM: Document Object Model
- HTML: Hypertext Markup Language
- HTTP: Hypertext Transfer Protocol

- JSON: JavaScript Object Notation
- PDF: Portable Document Format
- REST: REpresentational State Transfer
- SAX: Simple API for XML
- USP: Unique Selling Proposition
- W3CL World Wide Web Consortium
- WDDX: Web Distributed Data Exchange
- XML: Extensible Markup Language
- XML-DAS: XML Data Archiving Service
- XSL: EXtensible Stylesheet Language

Even the most hardened critic can agree that when it comes to working with XML-based technologies, you might do a lot worse than PHP. Very few programming languages offer the depth and breadth of XML extensions of PHP:

- SimpleXML and XMLReader for basic XML processing
- XSLT and XML-DAS for XML transformation
- SAX for event-based parsing
- DOM for tree-based parsing
- WDDX for data serialization
- XMLWriter for dynamic XML document creation

All of these extensions are based on the libxml2 library, and provide a robust, unified foundation for XML processing in PHP.

That said, one important XML-based technology doesn't appear in the list above: XQuery. While SimpleXML does include some basic XQuery support, it's not (nor is it supposed to be) a full-fledged XQuery processor. For that, you have to look further afield, to the PHP Extension Community Library (PECL), which contains an extension for the Zorba XQuery processor. This extension can be used to add full-featured XQuery processing support to PHP.

This article examines the Zorba PHP extension in detail, explains what it is and guides you through the process of installing and activating it for your PHP build. It provides detailed information on how to use this extension in the context of PHP scripts, explaining the Zorba API and illustrating it with various examples. It also examines some of Zorba's special functions for integrating with other Web technologies, such as REST and JSON.

Installing Zorba

First up, an introduction: Zorba is an XQuery processor,with a complete and fully compliant implementation of the W3C specifications for XQuery 1.0 and XQuery Update Facility 1.0. It also includes partial support for XQuery 1.1, an almost-complete implementation of the XQuery Scripting Extension 1.0, and various additional functions for e-mail transmission, PDF generation and code commenting. Zorba is currently available for Mac OS X, Linux®, and Microsoft® Windows® platforms as an open-source project under the terms of the Apache License, and it is supported by a number of organizations, including the FLWOR Foundation and Oracle Inc.

Zorba support in PHP comes through PECL's ext/zorba extension, which is maintained by William Candillon, and provides an object-oriented API for accessing the Zorba library. Although this extension is currently in alpha, it still allows you to do some fairly interesting things, including using FLWOR expressions, working with sequences and collections, updating and manipulating XML nodesets, and accessing Zorba-specific functionality through PHP. The extension is freely available under the PHP License.

To get started with ext/zorba, first make sure that you have a working PHP 5.3 installation, and then compile and install the Zorba libraries and PECL extension. Zorba requires a number of other packages to be pre-installed on the system for a clean compile so, before you start, make sure that you have recent versions of the following:

- libxml2
- iconv
- ICU (C version)
- Xerces (C version)
- SWIG (Simplified Wrapper and Interface Generator)
- GCC and related make/cmake utilities

You'll find links to download all of these packages, as well as Zorba itself (see Resources). Installation instructions will vary from package to package; in most cases, you'll need to follow the standard configure-make-install cycle, but you'll find more specific instructions in each package's documentation. Once you install them all, download the most recent release of Zorba (v1.2.0) and compile it for your system. If you're on a *NIX system with GCC v4.0.1 or better, you'll need these commands:

```
shell> tar -xzvf zorba-1.2.0.tar.gz
shell> cd zorba-1.2.0
shell> mkdir build
shell> cd build
shell> cmake -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

At this point, look at the output of cmake and verify that it includes messages similar to the following:

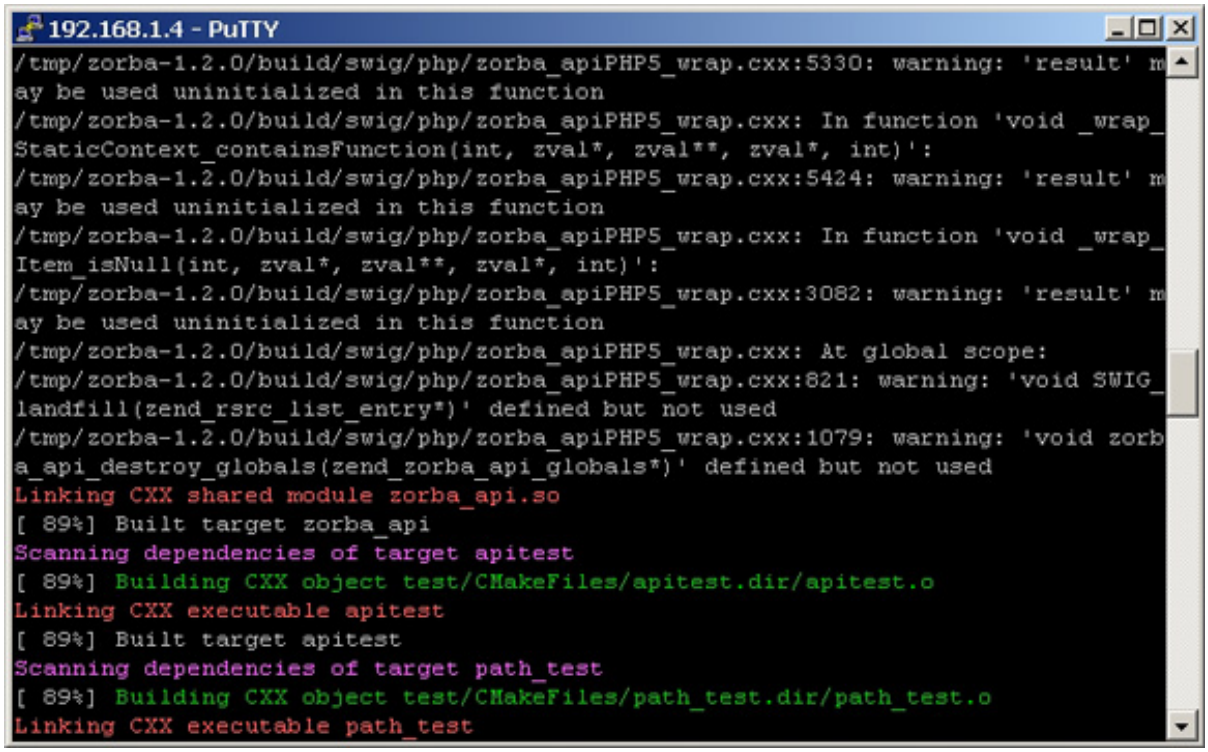
```
-- /usr/local/php/include/php/main
-- Found PHP5-Version 5.3.1

-- SWIG PHP: Generating PHP5 API
-- SWIG PHP: PHP module install path: /usr/local/php/lib/php/
   extensions/no-debug-non-zts-20090626
```

You can now go ahead and compile and install both Zorba and the PHP extension:

```
shell> make
shell> make install
```

At the end of this process, Zorba should be installed to the /usr/local/* branch of your file system. Figure 1 illustrates what you should see during the process.

Figure 1. Compiling Zorba and the Zorba PHP extension



To verify that it was correctly installed, open your command prompt, enter the

following command, and verify the resulting output:

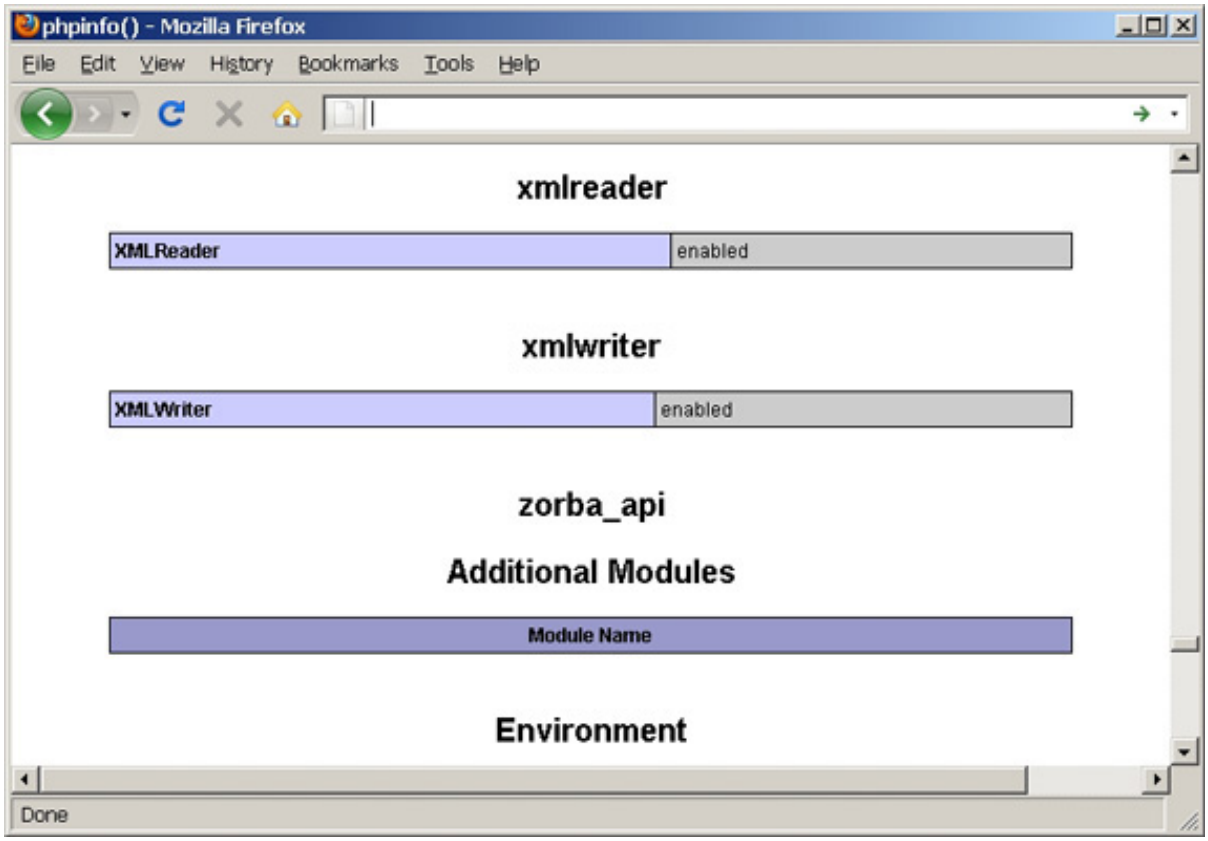
```
shell> zorba -q '40+1'
<?xml version="1.0" encoding="UTF-8"?>
41
```

At this point, you should also have a loadable PHP module named zorba_api.so in your PHP modules directory. You can check that the extension was correctly compiled with the following command:

```
shell> ctest -R php
Start processing tests
Test project /tmp/zorba-1.2.0/build
1188/1188 Testing php                                 Passed
100% tests passed, 0 tests failed out of 1
```

Enable the extension in the php.ini configuration file, restart the Web server, and check that the extension is active with a quick call to `phpinfo()`. Figure 2 illustrates what you should see.

Figure 2. Verifying the Zorba PHP extension



To wrap up the installation process, copy the zorba_api.php file from the source archive to a directory in your PHP include path, as below:

```
shell> cp zorba-1.2.0/zorba_api.php /usr/local/lib/php/
```

Understanding basic usage

Let's get started with a simple example. Take a look at Listing 1, which illustrates how to use Zorba to process XQuery queries from within a PHP script:

Listing 1. Assigning and displaying variables

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query string
  $queryStr = <<<END
  let \$message := 'She sells sea shells by the sea shore'
  return
  <result>{\$message}</result>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 2 displays the output of Listing 1:

Listing 2. The output of Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<result>She sells sea shells by the sea shore</result>
```

Listing 1 begins by including the Zorba PHP class definitions, and creating an instance of the memory store with the InMemoryStore class. Next, the `Zorba::getInstance()` method is used to initialize an instance of the Zorba processor; this instance serves as the primary access point to Zorba's XQuery functionality. Once these formalities are out of the way, the `compileQuery()` method is used to compile an XQuery, and the `execute()` method is used to

execute it.

What about the XQuery itself? It's a very simple example that makes use of the two most basic FLWOR components: the `let` clause, which assigns values to variables, and the `return` clause, which defines the structure of the XQuery result set. If you're familiar with XQuery, this should be quite familiar to you: The XQuery assigns a string value to a variable, and then returns it as an XML-encoded string. Notice the use of backslashes before XQuery variables within the PHP docblock; omitting these produces an error, because PHP tries to evaluate the variables instead of letting Zorba handle them.

Zorba supports declaring variables within the context of an XQuery, as well as basic math functions. Listing 3 demonstrates both these aspects:

Listing 3. Using math functions

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query string
  $queryStr = <<<END
  declare variable \$x := 30;
  declare variable \$y := 12;
  let \$r := \$x + \$y
  return
  <result> {\$r} </result>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 4 demonstrates the output of Listing 3:

Listing 4. The output of Listing 3

```
<?xml version="1.0" encoding="UTF-8"?>
<result>42</result>
```

Listing 3 uses the same basic template as Listing 1: initialize the Zorba class, define the query, and compile and execute it. In this case, the query first declares two variables, and then uses the `let` construct to add them together and the `result` construct to return the result.

Zorba supports the full range of XQuery functions. Consider Listing 5, which demonstrates the use of the `concat()` function in the context of an XQuery:

Listing 5. Concatenating strings

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query string
  $queryStr = <<<END
  declare variable \$x := 30;
  declare variable \$y := 12;
  let \$r := \$x + \$y
  return
  <result> {concat('The answer is: ', \$r)} </result>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 6 demonstrates the output:

Listing 6. The output of Listing 5

```
<?xml version="1.0" encoding="UTF-8"?>
<result>The answer is: 42</result>
```

Processing sequences

In addition to the let and return clauses, Zorba supports for sequences, as well as the `for` clause to process them. To illustrate this in practice, consider Listing 7:

Listing 7. Processing a sequence

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<<END
  let \$dolls := ('Echo', 'Victor', 'Sierra', 'November', 'Alpha')
  return
  <results>
  {
    for \$doll in \$dolls
    return
    <name>{\$doll}</name>
  }
  </results>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 7 defines a sequence of names, and then uses a `for` clause to loop through this sequence, retrieving each name and adding it to a node collection. Listing 7 also demonstrates how you can nest `return` statements to precisely control the result set generated by the query.

Listing 8 has the output of Listing 7:

Listing 8. The output of Listing 7

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
  <name>Echo</name>
  <name>Victor</name>
  <name>Sierra</name>
  <name>November</name>
  <name>Alpha</name>
</results>
```

You can also "implode" a sequence into a string with the XQuery `string-join()` function, as in Listing 9:

Listing 9. Concatenating sequence elements into a string

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<<END
  let \$dolls := ('Echo', 'Victor', 'Sierra', 'November', 'Alpha')
  return
  <results>
  {string-join(\$dolls, ', ')}
  </results>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 10 shows the output of Listing 9:

Listing 10. The output of Listing 9

```
<?xml version="1.0" encoding="UTF-8"?>
<results>Echo, Victor, Sierra, November, Alpha</results>
```

For generating numeric sequences, Zorba supports the `to` operator, which accepts the starting and ending value of a numeric range and generates a sequence containing all the numbers within that range. Consider Listing 11, which uses this to generate a sequence of the numbers between 1 and 20, and then uses the `mod` and `eq` operators to return a result set of only the even numbers:

Listing 11. Finding even numbers

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);
```

```
try {
  // create and compile query
  $queryStr = <<<END
  let \$set := (1 to 20)
  return
  <results>
  {
    for \$i in \$set
    where (\$i mod 2 eq 0)
    return
    <value> {\$i} </value>
  }
  </results>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 12 shows the output of Listing 11.

Listing 12. The output of Listing 11

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
  <value>2</value>
  <value>4</value>
  <value>6</value>
  <value>8</value>
  <value>10</value>
  <value>12</value>
  <value>14</value>
  <value>16</value>
  <value>18</value>
  <value>20</value>
</results>
```

Listing 11 also demonstrates Zorba's support for the XQuery `where` clause, which is commonly used to filter the output result set. More on this in the following section.

Filtering and sorting data

More often than not, you will deal with XML data originating from an external source—perhaps a flat file, perhaps a Web service, perhaps something else. To account for this, the Zorba PHP API includes an XmlDataManager class, which provides functions to load external XML documents into Zorba. Listing 13 displays a snippet of one such XML document, which you'll use as a base for the examples in

this section.

Listing 13. A sample XML file

```
<?xml version="1.0"?>
<data>
  <record>
    <code>ABW</code>
    <name>Aruba</name>
    <continent>North America</continent>
    <year></year>
    <population>103000</population>
    <gnp>828.00</gnp>
    <capital>
      <name>Oranjestad</name>
      <population>29034</population>
    </capital>
  </record>
  <record>
    <code>AFG</code>
    <name>Afghanistan</name>
    <continent>Asia</continent>
    <year>1919</year>
    <population>22720000</population>
    <gnp>5976.00</gnp>
    <capital>
      <name>Kabul</name>
      <population>1780000</population>
    </capital>
  </record>
  <record>
    <code>AUS</code>
    <name>Australia</name>
    <continent>Oceania</continent>
    <year>1901</year>
    <population>18886000</population>
    <gnp>351182.00</gnp>
    <capital>
      <name>Canberra</name>
      <population>322723</population>
    </capital>
  </record>
  <record>
    <code>AUT</code>
    <name>Austria</name>
    <continent>Europe</continent>
    <year>1918</year>
    <population>8091800</population>
    <gnp>211860.00</gnp>
    <capital>
      <name>Wien</name>
      <population>1608144</population>
    </capital>
  </record>
  <record>
    <code>AZE</code>
    <name>Azerbaijan</name>
    <continent>Asia</continent>
    <year>1991</year>
    <population>7734000</population>
    <gnp>4127.00</gnp>
    <capital>
      <name>Baku</name>
      <population>1787800</population>
    </capital>
  </record>
```

```
  ...
</data>
```

Listing 14 has an example of querying this document and retrieving a list of all the country names from it:

Listing 14. Retrieving country names

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // get XML data manager
  $dm = $zorba->getXmlDataManager();

  // read external XML document
  $dm->loadDocument('data.xml', file_get_contents('data.xml'));

  // create and compile query
  $queryStr = <<< END
  for \$record in doc('data.xml')//record
  let \$name := \$record/name/text()
  let \$code := \$record/code/text()
  return
  <name code="{\$code}">{\$name}</name>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 15 displays the output:

Listing 15. The output of Listing 14

```
<?xml version="1.0" encoding="UTF-8"?>
<name code="ABW">Aruba</name>
<name code="AFG">Afghanistan</name>
<name code="AGO">Angola</name>
<name code="AIA">Anguilla</name>
<name code="ALB">Albania</name>
<name code="AND">Andorra</name>
<name code="ANT">Netherlands Antilles</name>
<name code="ARE">United Arab Emirates</name>
<name code="ARG">Argentina</name>
```

```
<name code="ARM">Armenia</name>
<name code="ASM">American Samoa</name>
<name code="ATG">Antigua and Barbuda</name>
<name code="AUS">Australia</name>
<name code="AUT">Austria</name>
<name code="AZE">Azerbaijan</name>
<name code="BDI">Burundi</name>
<name code="BEL">Belgium</name>
<name code="BEN">Benin</name>
<name code="BFA">Burkina Faso</name>
<name code="BGD">Bangladesh</name>
...
```

Listing 16 updates Listing 14 to include a `where` clause that filters the result set to only those European countries with a population of more than 1000000, and an `order by` clause that sorts the result set in descending order of population:

Listing 16. Filtering countries by size and location

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // get XML data manager
  $dm = $zorba->getXmlDataManager();

  // read external XML document
  $dm->loadDocument('data.xml', file_get_contents('data.xml'));

  // create and compile query
  $queryStr = <<< END
  for \$record in doc('data.xml')//record
  let \$name := \$record/name/text()
  let \$code := \$record/code/text()
  let \$population := \$record/population/text()
  where contains(\$record/continent/text(), 'Europe')
  where (xs:integer(\$population) gt 1000000)
  order by \$population descending
  return
  <name code="{\$code}" population="{\$population}">{\$name}</name>
END;
  $query = $zorba->compileQuery($queryStr);

  // execute query and display result
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 17 displays the output of Listing 16:

Listing 17. The output of Listing 16

```
<?xml version="1.0" encoding="UTF-8"?>
<name code="AUT" population="8091800">Austria</name>
<name code="ALB" population="3401200">Albania</name>
<name code="BEL" population="10239000">Belgium</name>
```

Manipulating node collections

In addition to reading and querying XML, Zorba also includes functions that allow you to manipulate XML document trees by adding, updating, and deleting nodes. Listing 18 contains the simple XML file you'll use throughout this section:

Listing 18. An example XML document

```
<?xml version='1.0'?>
<heroes>
  <hero>
    <name>Spiderman</name>
    <alterego>Peter Parker</alterego>
  </hero>
  <hero>
    <name>Superman</name>
    <alterego>Clark Kent</alterego>
  </hero>
  <hero>
    <name>The Flash</name>
    <alterego>Wally West</alterego>
  </hero>
</heroes>
```

Listing 19 uses Zorba to read this file and add a new entry to the end using an XQuery Update expression:

Listing 19. Adding nodes

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // get XML data manager
  $dm = $zorba->getXmlDataManager();

  // read external XML document
  $dm->loadDocument('heroes.xml', file_get_contents('heroes.xml'));

  // create and compile query
  $queryStr = <<< END
  insert node
  <hero>
```

```
    <name>Batman</name>
    <alterego>Bruce Wayne</alterego>
  </hero>
  into doc('heroes.xml')//heroes
END;
  $query = $zorba->compileQuery($queryStr);
  $query->applyUpdates();

  // now check to see if the node was inserted
  $queryStr = <<< END
  for \$h in doc('heroes.xml')//heroes
  return
  <results> {\$h/hero} </results>
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 20 displays the revised XML document:

Listing 20. The output of Listing 19

```
<?xml version="1.0" encoding="UTF-8"?>
<heroes>
  <hero>
    <name>Spiderman</name>
    <alterego>Peter Parker</alterego>
  </hero>
  <hero>
    <name>Superman</name>
    <alterego>Clark Kent</alterego>
  </hero>
  <hero>
    <name>The Flash</name>
    <alterego>Wally West</alterego>
  </hero>
  <hero>
    <name>Batman</name>
    <alterego>Bruce Wayne</alterego>
  </hero>
</heroes>
```

You can also delete and replace nodes, as in Listing 21:

Listing 21. Deleting and replacing nodes

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
```

```
$zorba = Zorba::getInstance($ms);

try {
  // get XML data manager
  $dm = $zorba->getXmlDataManager();

  // read external XML document
  $dm->loadDocument('heroes.xml', file_get_contents('heroes.xml'));

  // create and compile query
  $queryStr = <<< END
  delete node doc('heroes.xml')//heroes/hero[last()],
  replace node doc('heroes.xml')//heroes/hero[last()]
  with
  <hero>
    <name>The Incredible Hulk</name>
    <alterego>Bruce Banner</alterego>
  </hero>
END;
  $query = $zorba->compileQuery($queryStr);
  $query->applyUpdates();

  // now check to see if the node was inserted
  $queryStr = <<< END
  doc('heroes.xml')
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 22 has the output of Listing 21:

Listing 22. The output of Listing 21

```
<?xml version="1.0" encoding="UTF-8"?>
<heroes>
  <hero>
    <name>Spiderman</name>
    <alterego>Peter Parker</alterego>
  </hero>
  <hero>
    <name>Superman</name>
    <alterego>Clark Kent</alterego>
  </hero>
  <hero>
    <name>The Incredible Hulk</name>
    <alterego>Bruce Banner</alterego>
  </hero>
</heroes>
```

Using REST functions

One of Zorba's strengths is its support for a wide variety of Web technologies and standards. This makes it suitable for use in a variety of "glue" applications that need XQuery support together with the capability to interact with data from varied external sources. For example, Zorba comes with a full-featured REST API, which allows developers to create, transmit, and process GET, POST, PUT, and DELETE requests and responses, all without leaving the Zorba environment.

Let's see how this works by using it with the del.icio.us REST API. Assuming you have an account with del.icio.us, you can make a GET request for http://user:pass@api.del.icio.us/v1/posts/recent and retrieve a list of recently-added bookmarks. The output is something like that in Listing 23:

Listing 23. Example REST output from the del.icio.us API

```
<?xml version='1.0' standalone='yes'?>
<posts tag="" user="someuser">
  <post href="http://www.kernel.org/"
   description="The Linux Kernel Archives"
   hash="7dae6d24e3f8c6c3c3aa1b05ce5bfe94"
   tag="linux kernel opensource" time="2007-12-04T09:03:25Z" />
  <post href="http://www.everythingphpmysql.com/"
   description="How to do Everything with PHP & MySQL - Vikram Vaswani"
   hash="8c9c6572c70cb3de3fa93b87269a8d79"
   tag="book development php mysql web beginner" time="2007-12-04T09:02:47Z" />
  <post href="http://www.mysql-tcr.com/"
   description="MySQL: The Complete Reference - Vikram Vaswani"
   hash="c8d99b00cfb9a1af9d59bbc6c46848cd"
   tag="mysql php book" time="2007-12-04T09:02:21Z" />
  ...
</posts>
```

If you do this with Zorba, you can further process the returned XML using XQuery. Consider Listing 24, which illustrates by performing the same GET request through Zorba, filtering the result set to only include those results tagged with 'book', and massaging the results into a different format from the original:

Listing 24. Performing a GET request

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<< END
  import module namespace zorba-rest =
    "http://www.zorba-xquery.com/zorba/rest-functions";
  for \$post in
    zorba-rest:get("http://user:pass@api.del.icio.us/v1/posts/recent")//posts/post
  where contains(\$post/@tag, 'book')
  return
  <result>
```

```
    <href> {string(\$post/@href)} </href>
    <description> {string(\$post/@description)} </description>
    <tags> {string(\$post/@tag)} </tags>
  </result>
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 25 displays the result:

Listing 25. The output of Listing 24

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <href>http://www.everythingphpmysql.com/</href>
  <description>How to do Everything with PHP & MySQL - Vikram Vaswani</description>
  <tags>book development php mysql web beginner</tags>
</result>
<result>
  <href>http://www.mysql-tcr.com/</href>
  <description>MySQL: The Complete Reference - Vikram Vaswani</description>
  <tags>mysql php book</tags>
</result>
```

In a similar vein, Zorba also offers `put()`, `post()`, `head()`, and `delete()` functions, which correspond to the PUT, POST, HEAD, and DELETE HTTP commands respectively. To add request parameters to these commands, specify them in the second argument, as in Listing 26:

Listing 26. Performing a POST request

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<< END
  import module namespace zorba-rest =
    "http://www.zorba-xquery.com/zorba/rest-functions";
  zorba-rest:post(
    'http://server/admin/login',
    <payload>
      <part name='username'>john</part>
      <part name='password'>guessme</part>
```

```
    </payload>
  )
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

The response from an HTTP transaction is automatically encoded into an XML result packet by Zorba. Listing 27 displays an example:

Listing 27. An encoded HTTP response packet

```
<?xml version="1.0" encoding="UTF-8"?>
<zorba-rest:result xmlns:zorba-rest="http://www.zorba-xquery.com/zorba/rest-functions">
  <zorba-rest:status-code>200</zorba-rest:status-code>
  <zorba-rest:headers>
    <zorba-rest:header zorba-rest:name="Date">Fri, 16 Oct 2009 12:20:15 GMT
    </zorba-rest:header>
    <zorba-rest:header zorba-rest:name="Server">
    Apache/2.2.11 (Win32) PHP/5.2.6
    </zorba-rest:header>
    <zorba-rest:header zorba-rest:name="X-Powered-By">PHP/5.2.6
    </zorba-rest:header>
    <zorba-rest:header zorba-rest:name="Content-Length">22
    </zorba-rest:header>
    <zorba-rest:header zorba-rest:name="Content-Type">text/html
    </zorba-rest:header>
  </zorba-rest:headers>
  <zorba-rest:payload>
    <result>1</result>
  </zorba-rest:payload>
</zorba-rest:result>
```

Using JSON functions

Why stop with REST? Zorba also includes functions to parse JSON-encoded data, and return it in an XML format. Listing 28 has an example of using Zorba's `json()` function to return an XML representation of a JSON packet:

Listing 28. Parsing JSON into XML

```
<?php
// set JSON data
$json = '{
  "books": [{
    "title":"Phantom Prey",
    "author":"John Sandford",
```

```
    "price":"7.99"
  }, {
    "title":"A Most Wanted Man",
    "author":"John le Carre",
    "price":"8.99"
  }, {
    "title":"Her Fearful Symmetry",
    "author":"Audrey Niffenegger",
    "price":"17.99"
  }, {
    "title":"The Watchman",
    "author":"Robert Crais",
    "price":"7.99"
  }, {
    "title":"Beach Babylon",
    "author":"Imogen Edward-Jones",
    "price":"12.99"
  }]}';

// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<< END
  import module namespace json =
    "http://www.zorba-xquery.com/zorba/json-functions";
  json:parse('{$json}')
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 29 displays the output of Listing 28:

Listing 29. The output of Listing 28

```
<?xml version="1.0" encoding="UTF-8"?>
<json type="object">
  <pair name="books" type="array">
    <item type="object">
      <pair name="title" type="string">Phantom Prey</pair>
      <pair name="author" type="string">John Sandford</pair>
      <pair name="price" type="string">7.99</pair>
    </item>
    <item type="object">
      <pair name="title" type="string">A Most Wanted Man</pair>
      <pair name="author" type="string">John le Carre</pair>
      <pair name="price" type="string">8.99</pair>
    </item>
    <item type="object">
      <pair name="title" type="string">Her Fearful Symmetry</pair>
```

```
      <pair name="author" type="string">Audrey Niffenegger</pair>
      <pair name="price" type="string">17.99</pair>
    </item>
    <item type="object">
      <pair name="title" type="string">The Watchman</pair>
      <pair name="author" type="string">Robert Crais</pair>
      <pair name="price" type="string">7.99</pair>
    </item>
    <item type="object">
      <pair name="title" type="string">Beach Babylon</pair>
      <pair name="author" type="string">Imogen Edward-Jones</pair>
      <pair name="price" type="string">12.99</pair>
    </item>
  </pair>
</json>
```

This is extremely useful, because it allows you to apply the power of XQuery to quickly extract the information you need from a JSON-encoded result packet. Listing 30 expands upon Listing 28, illustrating this in action with an XQuery to return the titles of all books under $10.00:

Listing 30. Parsing and filtering JSON with XQuery

```
<?php
// set JSON data
$json = '{
  "books": [{
    "title":"Phantom Prey",
    "author":"John Sandford",
    "price":"7.99"
  }, {
    "title":"A Most Wanted Man",
    "author":"John le Carre",
    "price":"8.99"
  }, {
    "title":"Her Fearful Symmetry",
    "author":"Audrey Niffenegger",
    "price":"17.99"
  }, {
    "title":"The Watchman",
    "author":"Robert Crais",
    "price":"7.99"
  }, {
    "title":"Beach Babylon",
    "author":"Imogen Edward-Jones",
    "price":"12.99"
  }]}';

// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<< END
  import module namespace json =
    "http://www.zorba-xquery.com/zorba/json-functions";
  let \$x := json:parse('{$json}')
  for \$i in \$x//item
  where (xs:decimal(\$i/pair[@name='price']/text()) lt 10.00)
```

```
  return
  <result> {\$i/pair[@name='title']/text()} </result>
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 31 displays the revised output:

Listing 31. The output of Listing 30

```
<?xml version="1.0" encoding="UTF-8"?>
<result>Phantom Prey</result>
<result>A Most Wanted Man</result>
<result>The Watchman</result>
```

You can use a number of other Zorba-specific utility functions within the PHP environment. For example, Listing 32 demonstrates the `random()` and `uuid()` functions, which produces a random integer and unique identifier respectively:

Listing 32. Generating unique and random identifiers

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // create and compile query
  $queryStr = <<< END
  import module namespace util =
    "http://www.zorba-xquery.com/zorba/util-functions";
  let \$uuid := util:uuid()
  let \$random := util:random()
  return
  <results>
    <uuid> {\$uuid} </uuid>
    <random> {\$random} </random>
  </results>
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
```

```
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 33 displays the output of Listing 32:

Listing 33. The output of Listing 32

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
 <uuid>87f0cf54-ba41-11de-a89c-9300a64ac3cd</uuid>
  <random>71869107</random>
</results>
```

Another example is the `serialize-to-string()` function, which accepts a collection of nodes and serializes them into a single string. Listing 34 has an example:

Listing 34. Serializing nodes

```
<?php
// include Zorba API
require_once 'zorba_api.php';

// create Zorba instance in memory
$ms = InMemoryStore::getInstance();
$zorba = Zorba::getInstance($ms);

try {
  // get XML data manager
  $dm = $zorba->getXmlDataManager();

  // read external XML document
  $dm->loadDocument('heroes.xml', file_get_contents('heroes.xml'));

  // create and compile query
  $queryStr = <<< END
  import module namespace util =
    "http://www.zorba-xquery.com/zorba/util-functions";
  let \$xml := doc('heroes.xml')//heroes/hero[1]
  let \$str:= util:serialize-to-string(\$xml)
  return
  <result> {\$str} </result>
END;
  $query = $zorba->compileQuery($queryStr);
  $result = $query->execute();
  echo $result;

  // clean up
  $query->destroy();
  $zorba->shutdown();
  InMemoryStore::shutdown($ms);
} catch (Exception $e) {
  die('ERROR:' . $e->getMessage());
}
?>
```

Listing 35 displays the output of Listing 34:

Listing 35. The output of Listing 34

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <hero>
    <name>Spiderman</name>
    <alterego>Peter Parker</alterego>
  </hero>
</result>
```

You can also use Zorba's built-in functions to send e-mail and create PDF documents. Read more about these functions in the Zorba documentation.

Conclusion

That's about it for this article. Over the last few pages, I gave you a crash course in how to add industrial-strength XQuery support to your PHP build with the Zorba XQuery processor. In addition to stepping you through the installation process, I showed you how to use Zorba to process basic FLWOR expressions, iterate over sequences, and read XML data from external sources. I also took you on a quick tour of Zorba's XQuery Update support, showing you how to dynamically add, update, and delete nodes from an XML document, and showed you how to exploit Zorba's USP—support for common glue technologies like REST and JSON—for maximum advantage in building Web applications.

Hopefully, all of this has served to whet your appetite, and encouraged you to give Zorba a try the next time you sit down to build a PHP/XML/XQuery application. Good luck, and happy coding!

Resources

Learn

- XQuery and FLWOR expressions: Learn the basics by example.
- XPath, XQuery and XQuery Update: Review the specifications on the W3C Web site.
- The official Zorba Web site and the Zorba documentation: Learn more about this general purpose XQuery processor.
- REST functionality and JSON support in Zorba: Read more about Zorba's support in the online documentation.
- A live demo of Zorba: Try some sample queries with this demo.
- the Zorba wiki and the Zorba bug-tracker: Participate in the Zorba community.
- The XML area on developerWorks: Get the resources you need to advance your skills in the XML arena.
- IBM XML certification: Find out how you can become an IBM-Certified Developer in XML and related technologies.
- XML technical library: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.

Get products and technologies

- The Zorba XQuery processor: Download this general purpose XQuery processor for use one many platforms.
- The Zorba PHP extension: Get a wrapper of Zorba library that allows PHP developers to use XQuery.
- The libxml2 package: Download the XML C parser and toolkit developed for the Gnome project.
- The iconv Unicode conversion library: Download this tool to convert between internal string representation (Unicode) and external string representation (a traditional encoding) during I/O.
- The ICU Unicode libraries: Download this set of C/C++ and Java libraries for Unicode and globalization support in your software applications.

- The Xerces XML parser: Download this validating XML parser and give your apps the ability to read and write XML data.
- The SWIG toolkit: Download this toolkit that allows you to connect high-level programming languages together.
- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- XML zone discussion forums: Participate in any of several XML-related discussions.
- developerWorks blogs: Check out these blogs and get involved in the developerWorks community.

About the author

Vikram Vaswani

Vikram Vaswani is the founder and CEO of Melonfire, a consulting services firm with special expertise in open-source tools and technologies. He is also the author of the books *PHP Programming Solutions* and *PHP: A Beginners Guide*.

Trademarks

in the United States, other countries, or both.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Other company, product, or service names may be trademarks or service marks of others.