PHP 101 (part 15): No News is Good News

A Difficult Choice

After the workout I gave you last time, you're probably either chomping at the bit to build another PHP application or you've decided to give up PHP programming
and try growing cucumbers instead. If it's the latter, you should stop reading right now, because I can guarantee you that this concluding installment of PHP 101 has absolutely nothing to teach you about vegetable farming.
If it's the former, however, then you're going to enjoy what's coming up. Over the next few pages, I'm going to be building a simple **RSS news aggregator** using PHP, SQLite and SimpleXML. With this news aggregator, you can plug into RSS news
feeds from all over the web, creating a newscast that reflects your needs and interests for your website. The best part: it updates itself automatically with the latest stories every time you view it!
Come on in, and let's get this show on the road!

Alphabet Soup

I'll start with the basics. What the heck is RSS anyhow?
RSS (the acronym stands for RDF Site Summary) is a format originally devised by Netscape to distribute information about the content on its My.Netscape.Com portal. The format has gone through many iterations since its introduction in early 1997 (take a look at http://backend.userland.com/stories/rss091 for information on RSS's
long and complicated history) but most feeds use RSS 1.0 or RSS 0.91, both of which are lightweight yet full-featured.

RSS makes it possible for webmasters to publish and distribute information about what's new and interesting on a particular site at a particular time. This information, which could range from a list of news articles to stock market data or weather forecasts, is published as a well-formed XML document, and can therefore be parsed, processed and rendered by any XML parser – including the SimpleXML parser that is part of PHP 5.

Quite a few popular web sites make an RSS or RDF news feed available to the public at large. Freshmeat and Slashdot both have one, and so do many others, including the PEAR, PECL and Zend sites. A quick Google search for **public RSS feeds** will get you more links than you can shake a stick at.

An RSS document typically contains a list of resources (URLs), marked up with descriptive metadata. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns="http://purl.org/rss/1.0/">
<channel rdf:about="http://www.melonfire.com/">
  <title>Trog</title>
  <description>Well-written technical articles and tutorials on web
technologies</description>
  <link>http://www.melonfire.com/community/columns/trog/</link>
  <items>
   <rdf:Seq>
```

```
      <li rdf:resource="http://www.melonfire.com/community/columns/trog/article.php?
id=100" />
   <li rdf:resource="http://www.melonfire.com/community/columns/trog/article.php?
id=71" />
   <li rdf:resource="http://www.melonfire.com/community/columns/trog/article.php?
id=62" />
  </rdf:Seq>
 </items>
</channel>
<item rdf:about="http://www.melonfire.com/community/columns/trog/article.php?id=100">
 <title>Building A PHP-Based Mail Client (part 1)</title>
  <link>http://www.melonfire.com/community/columns/trog/article.php?id=100</link>
 <description>Ever wondered how web-based mail clients work? Find out
here.</description>
</item>
<item rdf:about="http://www.melonfire.com/community/columns/trog/article.php?id=71">
 <title>Using PHP With XML (part 1)</title>
 <link>http://www.melonfire.com/community/columns/trog/article.php?id=71</link>
  <description>Use PHP's SAX parser to parse XML data and generate HTML
pages.</description>
</item>
<item rdf:about="http://www.melonfire.com/community/columns/trog/article.php?id=62">
 <title>Access Granted</title>
 <link>http://www.melonfire.com/community/columns/trog/article.php?id=62</link>
 <description>Precisely control access to information with the SQLite grant
```

```
tables.</description>

</item>

</rdf:RDF>
```

As you can see, an **RDF file** is split up into clearly demarcated sections. First comes the document **prolog**, **namespace declarations**, and **root element**.

This is followed by a `<channel>` block, which contains general information on the **channel** that is described by this RDF file. In the example above, the channel is Melonfire's Trog column, which gets updated every week with new technical articles and tutorials.

The `<channel>` block contains an `<items>` block, which contains a sequential list of all the resources described within the RDF document.

Every resource in this block corresponds to a resource described in greater detail in a subsequent `<item>` block. Every `<item>` block describes a single resource in greater detail, providing a title, an URL and a description of that resource. It's this information that our application will use to generate a personalized news feed.

Laying the Foundation

Now that you know what RSS and RDF are all about, it's time to start work. I'll begin by

sitting down at a table near the window and doodling aimlessly on a sheet of paper until
I figure out exactly what my application is supposed to do, piece by piece (actually, in
this case, the **requirements** are actually pretty basic):

1.The application must support one or more RSS-compliant news feeds. On start-up, the
application should retrieve the latest versions of these feeds, parse them and display
their contents in an easy-to-read manner. A SQLite database is a good choice to store
this list of feeds.
2.The user should be able to control the number of stories s/he picks up from each feed. For example, a user might want to display more science and health news than
business news.
3.The application should offer the user a web-based interface to add or delete news
feeds. This interface will use PHP's SQLite API to run appropriate SQL queries on the
SQLite database file and alter the information stored in the database.

Keeping these requirements in mind, it's possible to **design a simple database** table to hold the (user-configurable) list of RSS news feeds. Here's what it might look like:

```
CREATE TABLE rss (
 id INTEGER NOT NULL PRIMARY KEY,
 title varchar(255) NOT NULL,
 url varchar(255) NOT NULL,
 count INTEGER NOT NULL
);
```

From the table above, it's clear that every news feed will have three attributes: a descriptive title, the URL to the feed itself, and a value indicating how many of the stories in the feed you would like to see displayed in your own custom news page. Let's add some data to get things started:

```
INSERT INTO rss VALUES(1, 'Slashdot',
'http://slashdot.org/slashdot.rdf', 5);
INSERT INTO rss VALUES(2, 'Wired News',
'http://www.wired.com/news_drop/netcenter/netcenter.rdf', 5);
INSERT INTO rss VALUES(3, 'Business News',
'http://www.npr.org/rss/rss.php?topicId=6', 3);
INSERT INTO rss VALUES(4, 'Health News',
'http://news.bbc.co.uk/rss/newsonline_world_editio
n/health/rss091.xml', 3);
INSERT INTO rss VALUES(5, 'Freshmeat',
'http://www.freshmeat.net/backend/fm-releases.rdf', 5);
```

You can create all this directly from the **schema** file *rss.sql* using the SQLite command `.read` from the command-line client, if you still have that on board from Part Nine. In fact, now would be a good time for you to download all the source code for this application, so that you can check it out and refer to it easily throughout this tutorial. Note that you will need a PHP 5-enabled web server to run this code.

Top Story

With the database safely in its web-inaccessible directory, the next step is to write the code that uses the data inside it to connect to each news feed, parse it for news data, and present a customized news page.

Here's what that code, *user.php*, looks like:

```
<?php
```

```
// PHP 5
// include configuration file
include('config.php');
// open database file
$handle = sqlite_open($db) or die('ERROR: Unable to open database!');
// generate and execute query
$query = "SELECT id, title, url, count FROM rss";
$result = sqlite_query($handle, $query) or die("ERROR: $query.
".sqlite_error_string(sqlite_last_error($handle)));
// if records present
if (sqlite_num_rows($result) > 0) {
        // iterate through resultset
    // fetch and parse feed
        while($row = sqlite_fetch_object($result)) {
                $xml = simplexml_load_file($row->url);
                echo "<h4>$row->title</h4>";
        // print descriptions
        for ($x = 0; $x < $row->count; $x++) {
                        // for RSS 0.91
            if (isset($xml->channel->item)) {
                $item = $xml->channel->item[$x];
                        }
            // for RSS 1.0
            elseif (isset($xml->item)) {
                $item = $xml->item[$x];
                        }
            echo "<a href=\"$item->link\">$item->title</a><br />$item->description<p />";
        }
                echo "<hr />";
                // reset variables
```

```
        unset($xml);
        unset($item);
         }
}
// if no records present
// display message
else {
?>

    <font size = '-1'>No feeds currently configured</font>
<?php
}
// close connection
sqlite_close($handle);
?>
```

Here’s what the output might look like (note that there will a time lag in producing the page, because PHP will be silently opening HTTP connections to each URL to retrieve
the corresponding RSS feed):

The code to accomplish this might look simple, but there’s actually a lot going on behind
the scenes. The first step is to obtain a list of the RSS feeds configured by the user from the SQLite database. To accomplish this, a SQLite database handle is initialized, and
a SQL `SELECT` query is executed. A `while()` loop is used to iterate
through the resulting record collection.

For each URL thus obtained, the `simplexml_load_file()` function is used to retrieve and read the RSS feed. Depending on the number of stories to be displayed, a `for()` loop is executed and the appropriate number of `<item>` elements in the feed are parsed. Notice that the path to access an `<item>` differs depending on whether the feed is RSS 0.91 or RSS 1.0.

Note that if the database is empty, an error message will appear. In this example, since I've already inserted a bunch of records into the database, you'll never see the error message at all; however, it's good programming practice to ensure that all eventualities are accounted for, even remote ones.

As before, the file *config.php* is included at the top of every script. This file contains database access parameters, as below:

```
<?php
// database details
// always use a directory that cannot be accessed from the web
$path = $_SERVER['DOCUMENT_ROOT'].'/../';
$db = $path.'rss.db';
?>
```

Point and Click

With the news display out of the way, all that's left is to add a simple administrative tool to manipulate the contents of the SQLite database. The code here is going to be

very similar to what you saw in PHP 101 Part 14: a start page called *admin.php* that provides a snapshot of the current database, and a form to add new entries. Here it is in full:

```
<html>
<head><basefont face = 'Arial'></head>
<body>
<h2>Feed Manager</h2>
<h4>Current Feeds:</h4>
<table border = '0' cellspacing = '10'>
<?php
// PHP 5
// include configuration file
include('config.php');
// open database file
$handle = sqlite_open($db) or die('ERROR: Unable to open database!');
// generate and execute query
$query = "SELECT id, title, url, count FROM rss";
$result = sqlite_query($handle, $query) or die("ERROR: $query.
".sqlite_error_string(sqlite_last_error($handle)));
// if records present
if (sqlite_num_rows($result) > 0) {
        // iterate through result set
    // print article titles
        while ($row = sqlite_fetch_object($result)) {
    ?>
```

```
        <tr>
        <td><?php echo $row->title; ?> (<?php echo $row->count; ?>)</td>
                <td><font size = '-2'><a href="delete.php?id=<?php echo $row->id; ?
>">delete</a></font></td>
        </tr>
        <?php
    }
}
// if there are no records present, display message
else {
?>

    <font size = '-1'>No feeds currently configured</font>
<?php
}
// close connection
sqlite_close($handle);
?>
</table>
<h4>Add New Feed:</h4>
<form action = 'add.php' method = 'post'>
<table border = '0' cellspacing = '5'>
<tr>
    <td>Title</td>
        <td><input type = 'text' name = 'title'></td>
</tr>
<tr>
```

```
    <td>Feed URL</td>
    <td><input type = 'text' name = 'url'></td>
</tr>
<tr>
    <td>Stories to display</td>
    <td><input type = 'text' name = 'count' size = '2'></td>
</tr>
<tr>
    <td colspan = '2' align = 'right'><input type = 'submit' name = 'submit' value = 'Add RSS
Feed'></td>
</tr>
</table>
</form>
</body>
</html>
```

Here’s what it looks like:

As you can see, there are two sections in this script. The first half connects to the database and prints a list of all the currently configured news feeds, with a “delete” link next to each. The second half contains a form for the administrator to add a new feed, together with its attributes.

Once the form is submitted, the data gets `POST`-ed to the script *add.php*, which validates it and saves it to the database. Here’s the code for

add.php:

```
<html>
<head><basefont face = 'Arial'></head>
<body>
<h2>Feed Manager</h2>
<?php
// PHP 5
if (isset($_POST['submit'])) {
    // check form input for errors

    // check title
    if (trim($_POST['title']) == '') {
            die('ERROR: Please enter a title');
    }
        // check URL
    if ((trim($_POST['url']) == '') || !ereg("^http\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(:[a-zA-Z0-
9]*)?/?([a-zA-Z0-9\._\?\,\'/\\\+&%\$#\=~\-])*$",$_POST['url'])) {
            die('ERROR: Please enter a valid URL');
    }
        // check story number
    if (!is_numeric($_POST['count'])) {
            die('ERROR: Please enter a valid story count');
    }
        // include configuration file
    include('config.php');
        // open database file
    $handle = sqlite_open($db) or die('ERROR: Unable to open database!');

```

```
    // generate and execute query
    $query = "INSERT INTO rss (title, url, count) VALUES ('".$_POST['title']."', '".$_POST['url']."',
'".$_POST['count']."')";
        $result = sqlite_query($handle, $query) or die("ERROR: $query.
".sqlite_error_string(sqlite_last_error($handle)));
        // close connection
    sqlite_close($handle);
        // print success message
    echo "Item successfully added to the database! Click <a href = 'admin.php'>here</a> to
return to the main page";
}
else {
    die('ERROR: Data not correctly submitted');
}
?>


</body>
</html>
```

The lower half of the script should be familiar to you: it contains the usual function calls to open an SQLite database and execute an `INSERT` query to save the user's data to the database. What's interesting, though, is the top half of the script,
which contains a number of input tests to ensure that the data being saved doesn't
contain gibberish.

There are three tests here. One checks for the presence of a descriptive title,

another

uses the `is_numeric()` function to verify that the value entered for the

story count is a valid number, and the third uses the `ereg()` function to

check the format of the URL. If you read Part 13, you'll

know all about the importance of validating user input; here's that theory going

into

action.

That takes care of adding new RSS feeds. Now, what about removing them? Remember how, in *admin.php*, each feed displayed in the list had a "delete" link, which pointed to the script *delete.php*. This *delete.php* script takes care of deleting a news feed from the table, given the feed ID (which is passed through the link). Take a look at the code, and things will become clearer:

```
<html>
<head><basefont face = 'Arial'></head>
<body>
<h2>Feed Manager</h2>
<?php
// PHP 5
if (isset($_GET['id']) && is_numeric($_GET['id'])) {
    // include configuration file
    include('config.php');

    // open database file
    $handle = sqlite_open($db) or die('ERROR: Unable to open database!');
```

```
    // generate and execute query
    $query = "DELETE FROM rss WHERE id = '".$_GET['id']."'";
    $result = sqlite_query($handle, $query) or die("ERROR: $query.
".sqlite_error_string(sqlite_last_error($handle)));

    // close connection
    sqlite_close($handle);
    // print success message
    echo "Item successfully removed from the database! Click <a href =
'admin.php'>here</a> to return to the main page";
}
else {
    die('ERROR: Data not correctly submitted');
}
?>


</body>
</html>
```

The record ID passed through the URL `GET` method is retrieved by *delete.php*, and used with a `DELETE` SQL query to erase the corresponding record. Try it out and see for yourself!

And that’s about all I have for you. I hope you enjoyed the 15-part journey that was PHP 101,
and that you found it both educational and fun – I know I did, and it was a pleasure having
you along for the ride. If you’d like to read more about specific aspects of PHP,

drop by

www.melonfire.com and take a look at some more of my tutorials and articles.

Until then... happy coding!