By icarus

Table of Contents

Fame And Immense Fortune

Imagine a site that carried the latest news from the hottest portals on the Web. Stock prices, weather information, news stories, threaded discussions, software releases...all updated dynamically, on an hourly basis, without any manual intervention. Imagine the traffic such a site would get. Imagine the advertising dollars that would pour in, and the adulation that the webmaster would receive.

Now stop imagining and start reading, because, if you pay close attention, that webmaster could be you and that site could be yours. All you need is a little imagination, some clever PHP coding and a couple of free RSS files. And, obviously, the remaining nine pages of this article.

Have Content, Will Syndicate

We'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 the end of this article for links to information on RSS's long and complicated history − but seems to have stabilized at RSS 1.0, an RDF−compliant version that is both lightweight and 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 point in 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.

By making it possible to distribute a frequently−updated list of the latest information about a particular Web site, RSS opens the door to simple, easy content syndication over the Web. In order to understand how, consider the following simple example:

Site A, a news site (a "content syndicator"), could publish, on a hourly basis, an RSS document containing a list of the latest news from around the world, together with links to the full article on the site. This RSS document could be picked up by another Web site (Site B, a "content aggregator"), parsed, and displayed on Site B's index page. Every time Site A publishes a new RSS document, Site B's index page automatically gets updated with the latest news.

This kind of arrangement works for both organizations involved in the transaction. Since the links within the RSS document all point to articles on Site A, Site A will immediately experience an increase in visitors and site traffic. And Site B's webmaster can take the week off, since he now has a way to automagically update his index page, simply by linking it to the dynamic content being published by Site A.

Quite a few popular Web sites make an RSS or RDF news feed available to the public at large. Freshmeat, (http://www.freshmeat.net) and Slashdot (http://www.slashdot.org) both have one, and so do many others. I'll be using Freshmeat's RDF file extensively over the course of this article; however, the techniques described here can be applied to any RSS 1.0 or RDF file.

Switching Channels

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.ph
p?id
=100" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=71" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=62" />
</rdf:Seq>
</items>
</channel>

<item
rdf:about="http://www.melonfire.com/community/columns/trog/article.php?i
d=10
0">
<title>Building A PHP-Based Mail Client (part 1)</title>

<link>http://www.melonfire.com/community/columns/trog/article.php?id=100
</li
nk>
<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?i
d=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?i
d=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 mySQL grant tables.</description>
</item>

</rdf:RDF>
```

As you can see, an RDF file is split up into very clearly−demarcated sections. First comes the document prolog,

```
<?xml version="1.0" encoding="UTF−8"?>
```

and namespace declarations in the root element.

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xmlns="http://purl.org/rss/1.0/">
```

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.

```
<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.ph
p?id
=100" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=71" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=62" />
</rdf:Seq>
</items>
</channel>
```

The <channel> block contains an <items> block, which contains a sequential list of all the resources described within the RDF document. This list is formulated as a series of elements, which may be familiar to you from HTML. Every resource in this block corresponds to a resource described in greater detail in a subsequent <item> block (further down).

```
<items>
<rdf:Seq>
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=100" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=71" />
<li
rdf:resource="http://www.melonfire.com/community/columns/trog/article.ph
p?id
=62" />
</rdf:Seq>
</items>
```

It's also possible to intersperse an <image> block at this stage, in case you'd like to publish the URL of your channel's logo.

And now for the meat. Every <item> block in an RSS 1.0 document describes a single resource in greater

detail, providing a title, a URL and a description of that resource.

```
<item
rdf:about="http://www.melonfire.com/community/columns/trog/article.php?i
d=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>
```

In this example, the <item> block above describes a single article in the Trog "channel", providing a description and title for it, and including a URL so that a content aggregator can create backward links to it.

As you can see, an RSS 1.0 file is fairly straightforward, and it's quite easy to create one, either manually or programmatically. The example and explanation above was merely illustrative; there's a lot more you can do with RSS 1.0 and RDF in general, and you should take a look at the links provided at the end of this article for more information. Until we get there, though, let's spend a few more minutes discussing how an RSS 1.0 document like the one above can be plugged into your own Web site.

Fresh Meat

Since RSS is, technically, well−formed XML, it can be processed using standard XML programming techniques. There are two primary techniques here: SAX (the Simple API for XML) and DOM (the Document Object Model).

A SAX parser works by traversing an XML document and calling specific functions as it encounters different types of tags. So, for example, I might call a specific function to process a starting tag, another function to process an ending tag, and a third function to process the data between them. The parser's responsibility is simply to sequentially traverse the document; the functions it calls are responsible for processing the tags found. Once the tag is processed, the parser moves on to the next element in the document, and the process repeats itself.

A DOM parser, on the other hand, works by reading the entire XML document into memory, converting it into a hierarchical tree structure, and then providing an API to access different tree nodes (and the content attached to each). Recursive processing, together with API functions that allow developers to distinguish between different types of nodes (element, attribute, character data, comment et al), make it possible to perform different actions depending on both node type and node depth within the tree.

SAX and DOM parsers are available for almost every language, including your favourite and mine, PHP. I'll be using PHP's SAX parser to process the RDF examples in this article; however, it's just as easy to perform equivalent functions using the DOM parser.

Let's look at a simple example to put all this in perspective. Here's the RDF file I'll be using, culled directly from http://www.freshmeat.net/ :

```
<?xml version="1.0" encoding="ISO−8859−1"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xmlns="http://purl.org/rss/1.0/"
xmlns:dc="http://purl.org/dc/elements/1.1/"
>
<channel rdf:about="http://freshmeat.net/">
<title>freshmeat.net</title>
<link>http://freshmeat.net/</link>
<description>freshmeat.net maintains the Web's largest index
of Unix
and cross−platform open source software. Thousands of
applications are
meticulously cataloged in the freshmeat.net database, and
links to new
code are added daily.</description>
<dc:language>en−us</dc:language>
<dc:subject>Technology</dc:subject>
<dc:publisher>freshmeat.net</dc:publisher>
<dc:creator>freshmeat.net contributors</dc:creator>
<dc:rights>Copyright (c) 1997−2002 OSDN</dc:rights>
```

```
<dc:date>2002-02-11T10:20+00:00</dc:date>
<items>
<rdf:Seq>
<rdf:li rdf:resource="http://freshmeat.net/releases/69583/" />
<rdf:li rdf:resource="http://freshmeat.net/releases/69581/" />

<!-- and so on ->

</rdf:Seq>
</items>
<image rdf:resource="http://freshmeat.net/img/fmII-button.gif"
/>
<textinput rdf:resource="http://freshmeat.net/search/" />
</channel>

<image rdf:about="http://freshmeat.net/img/fmII-button.gif">
<title>freshmeat.net</title>
<url>http://freshmeat.net/img/fmII-button.gif</url>
<link>http://freshmeat.net/</link>
</image>

<item rdf:about="http://freshmeat.net/releases/69583/">
<title>sloop.splitter 0.2.1</title>
<link>http://freshmeat.net/releases/69583/</link>
<description>A real time sound effects program.</description>
<dc:date>2002-02-11T04:52-06:00</dc:date>
</item>

<item rdf:about="http://freshmeat.net/releases/69581/">
<title>apacompile 1.9.9</title>
<link>http://freshmeat.net/releases/69581/</link>
<description>A full-featured Apache compilation
HOWTO.</description>
<dc:date>2002-02-11T04:52-06:00</dc:date>
</item>

<!-- and so on ->

</rdf:RDF>
```

And here's the PHP script to parse it and display the data within it:

```
<?php
// XML file
$file = "fm-releases.rdf";

// set up some variables for use by the parser
```

```
$currentTag = "";
$flag = "";

// create parser
$xp = xml_parser_create();

// set element handler
xml_set_element_handler($xp, "elementBegin", "elementEnd");
xml_set_character_data_handler($xp, "characterData");
xml_parser_set_option($xp, XML_OPTION_CASE_FOLDING, TRUE);

// read XML file
if (!($fp = fopen($file, "r")))
{
die("Could not read $file");
}

// parse data
while ($xml = fread($fp, 4096))
{
if (!xml_parse($xp, $xml, feof($fp)))
{
die("XML parser error: " .
xml_error_string(xml_get_error_code($xp)));
}
}

// destroy parser
xml_parser_free($xp);

// opening tag handler
function elementBegin($parser, $name, $attributes)
{
global $currentTag, $flag;
// export the name of the current tag to the global scope
$currentTag = $name;
// if within an item block, set a flag
if ($name == "ITEM")
{
$flag = 1;
}
}

// closing tag handler
function elementEnd($parser, $name)
{
global $currentTag, $flag;
$currentTag = "";
// if exiting an item block, print a line and reset the flag
```

```
if ($name == "ITEM")
{
echo "<hr>";
$flag = 0;
}
}

// character data handler
function characterData($parser, $data)
{
global $currentTag, $flag;
// if within an item block, print item data
if (($currentTag == "TITLE" || $currentTag == "LINK" ||
$currentTag ==
"DESCRIPTION") && $flag == 1)
{
echo "$currentTag: $data <br>";
}
}

?>
```

Don't get it yet? The next page has an explanation.

Capture The Flag

The first step in this script is to set up some global variables:

```
// XML file
$file = "fm-releases.rdf";

// set up some variables for use by the parser
$currentTag = "";
$flag = "";
```

The $currentTag variable will hold the name of the element that the parser is currently processing − you'll see why this is needed shortly.

Since my ultimate goal is to display the individual items in the channel, with links, I also need to know when the parser has exited the <channel>...</channel> block and entered the <item>...</item> sections of the document. Since I'm using a SAX parser, which operates in a sequential manner, there is no parser API available to discover depth or location in the tree. So I have to invent my own mechanism to do this − which is where the $flag variable comes in.

The $flag variable will be used to find out if the parser is within the <channel> block or the <item> block.

The next step is to initialize the SAX parser and begin parsing the RSS document.

```
// create parser
$xp = xml_parser_create();

// set element handler
xml_set_element_handler($xp, "elementBegin", "elementEnd");
xml_set_character_data_handler($xp, "characterData");
xml_parser_set_option($xp, XML_OPTION_CASE_FOLDING, TRUE);

// read XML file
if (!($fp = fopen($file, "r")))
{
die("Could not read $file");
}

// parse data
while ($xml = fread($fp, 4096))
{
if (!xml_parse($xp, $xml, feof($fp)))
{
die("XML parser error: " .
xml_error_string(xml_get_error_code($xp)));
```

```
}
}

// destroy parser
xml_parser_free($xp);
```

This is a fairly standard sequence of commands, and the comments should explain it sufficiently. The xml_parser_create() function is used to instantiate the parser, and assign it to the handle $xp. Next, callback functions are set up to handle opening and closing tags, and the character data within them. Finally, the xml_parse() function, in conjunction with a bunch of fread() calls, is used to read the RDF file and parse it.

Each time an opening tag is encountered in the document, the opening tag handler elementBegin() is called.

```
// opening tag handler
function elementBegin($parser, $name, $attributes)
{
global $currentTag, $flag;
// export the name of the current tag to the global scope
$currentTag = $name;
// if within an item block, set a flag
if ($name == "ITEM")
{
$flag = 1;
}
}
```

This function receives, as function argument, the name of the current tag and its attributes (if any). This tag name is assigned to the global $currentTag variable, and − if the tag is an opening <item> tag − the $flag variable is set to 1.

Conversely, when a closing tag is found, the closing tag handler elementEnd() is invoked.

```
// closing tag handler
function elementEnd($parser, $name)
{
global $currentTag, $flag;
$currentTag = "";
// if exiting an item block, print a line and reset the flag
if ($name == "ITEM")
{
echo "<hr>";
$flag = 0;
}
}
```

This closing tag handler also receives the tag name as parameter. If this is a closing </item> tag, the value of $flag is reset to 0, and the value of $currentTag is cleared.

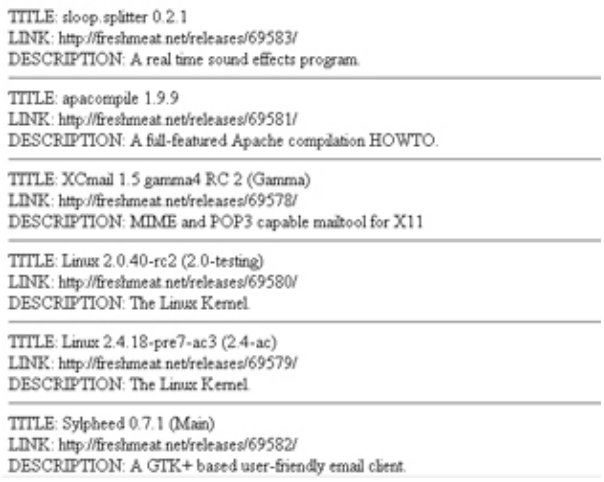
Now, what about the data between the tags, which is what we're really interested in? Say hello to the character data handler, characterData().

```
// character data handler
function characterData($parser, $data)
{
global $currentTag, $flag;
// if within an item block, print item data
if (($currentTag == "TITLE" || $currentTag == "LINK" ||
$currentTag ==
"DESCRIPTION") && $flag == 1)
{
echo "$currentTag: $data <br>";
}
}
```

Now, if you look at the arguments passed to this function, you'll see that characterData() only receives the data between the opening and closing tag − it has no idea which particular tag the parser is currently processing. Which is why we needed the global $currentTag variable in the first place (told you this would make sense eventually!)

If the value of $flag is 1 − in other words, if the parser is currently within an <item>...</item> block − and if the element currently being processed is either a <title>, <link> or <description> element, then the data is printed to the output device (in this case, the Web browser), followed by a line break.

The entire RDF document will be processed in this sequential manner, with output appearing every time an item is found. Here's what you'll see when you run the script:



Nesting Time

Now, the previous example was just fine for illustrative purposes − but if you're serious about plugging content into your Web site, you're going to need something a lot prettier. So take a look at this evolution of the previous script, which makes a few additions to simplify the task of formatting the RDF data.

```
<html>
<head>
<basefont face="Verdana">
</head>
<body>

<table border="0" cellspacing="5" cellpadding="5">
<tr>
<td><b>New releases on freshmeat.net today:</b></td>
</tr>

<?php
// XML file
$file = "http://www.freshmeat.net/backend/fm-releases.rdf";

// set up some variables for use by the parser
$currentTag = "";
$flag = "";
$count = 0;

// this is an associative array of channel data with keys
("title",
"link",
"description")
$channel = array();

// this is an array of arrays, with each array element
representing an
<item> // each outer array element is itself an associative
array
// with keys ("title", "link", "description")
$items = array();

// opening tag handler
function elementBegin($parser, $name, $attributes)
{
global $currentTag, $flag;
$currentTag = $name;
// set flag if entering <channel> or <item> block
if ($name == "ITEM")
{
```

```
$flag = 1;
}
else if ($name == "CHANNEL")
{
$flag = 2;
}
}

// closing tag handler
function elementEnd($parser, $name)
{
global $currentTag, $flag, $count;
$currentTag = "";

// set flag if exiting <channel> or <item> block
if ($name == "ITEM")
{
$count++;
$flag = 0;
}
else if ($name == "CHANNEL")
{
$flag = 0;
}
}

// character data handler
function characterData($parser, $data)
{
global $currentTag, $flag, $items, $count, $channel;
$data = trim(htmlspecialchars($data));
if ($currentTag == "TITLE" || $currentTag == "LINK" ||
$currentTag ==
"DESCRIPTION")
{
// add data to $channels[] or $items[] array
if ($flag == 1)
{
$items[$count][strtolower($currentTag)] .=
$data;
}
else if ($flag == 2)
{
$channel[strtolower($currentTag)] .= $data;
}
}

}
```

```
// create parser
$xp = xml_parser_create();

// set element handler
xml_set_element_handler($xp, "elementBegin", "elementEnd");
xml_set_character_data_handler($xp, "characterData");
xml_parser_set_option($xp, XML_OPTION_CASE_FOLDING, TRUE);
xml_parser_set_option($xp, XML_OPTION_SKIP_WHITE, TRUE);

// read XML file
if (!($fp = fopen($file, "r")))
{
die("Could not read $file");
}

// parse data
while ($xml = fread($fp, 4096))
{
if (!xml_parse($xp, $xml, feof($fp)))
{
die("XML parser error: " .
xml_error_string(xml_get_error_code($xp)));
}
}

// destroy parser
xml_parser_free($xp);

// now iterate through $items[] array
// and print each item as a table row
foreach ($items as $item)
{
echo "<tr><td><a href=" . $item["link"] . ">" . $item["title"]
.
"</a><br>" . $item["description"] . "</td></tr>"; }

?>
</table>
</body>
</html>
```

The major difference between this iteration and the previous one is that this one creates two arrays to hold the information extracted during the parsing process. The $channel array is an associative array which holds basic descriptive information about the channel being processed, while the $items array is a two−dimensional array containing information on the individual channel items. Every element of the $items array is itself an associative array, with named keys for the item title, URL and description; the total number of elements in $items is equal to the total number of <item> blocks in the RDF document.

Note also the change to the $flag variable, which now holds two values depending on whether a <channel>...</channel> or <item>...</item> block is being processed. This is necessary so that the parser puts the information into the appropriate array.

Once the document has been fully parsed, it's a simple matter to iterate through the $items array and print each item as a table row. Here's what the end result looks like:

Back To Class

Now, with all this power at your command, why on earth would you want to restrict yourself to just a single RDF feed? As I said earlier, most major sites make snapshots of their content available on a regular basis...and it's fairly simple to plug all these different feeds into your own site. Let's see how.

The first thing to do is modularize the code from the previous example, so that you don't need to repeat it over and over again for every single feed. I'm going to simplify things by packaging it into a class, and including the class file into my final PHP script.

Here's the class:

```
<?
class RDFParser
{
//
// variables
//

// set up local variables for this class
var $currentTag = "";
var $flag = "";
var $count = 0;

// this is an associative array of channel data with keys
("title", "link", "description")
var $channel = array();

// this is an array of arrays, with each array element
representing an <item>
// each outer array element is itself an associative array
// with keys ("title", "link", "description")
var $items = array();


//
// methods
//

// set the name of the RDF file to parse
// this is usually a local file
// you may set it to a remote file if your PHP build supports
URL fopen()
function setResource($file)
{
$this->file = $file;
}
```

```
// parse the RDF file set with setResource()
// this populates the $channel and $items arrays
function parseResource()
{
// create parser
$this->xp = xml_parser_create();

// set object reference
xml_set_object($this->xp, $this);

// set handlers and parser options
xml_set_element_handler($this->xp, "elementBegin",
"elementEnd");
xml_set_character_data_handler($this->xp,
"characterData");
xml_parser_set_option($this->xp,
XML_OPTION_CASE_FOLDING, TRUE);
xml_parser_set_option($this->xp, XML_OPTION_SKIP_WHITE,
TRUE);

// read XML file
if (!($fp = fopen($this->file, "r")))
{
die("Could not read $this->file");
}

// parse data
while ($xml = fread($fp, 4096))
{
if (!xml_parse($this->xp, $xml, feof($fp)))
{
die("XML parser error: " .
xml_error_string(xml_get_error_code($this->xp)));
}
}

// destroy parser
xml_parser_free($this->xp);
}

// opening tag handler
function elementBegin($parser, $name, $attributes)
{
$this->currentTag = $name;
// set flag if entering <channel> or <item> block
if ($name == "ITEM")
{
```

```
$this->flag = 1;
}
else if ($name == "CHANNEL")
{
$this->flag = 2;
}
}

// closing tag handler
function elementEnd($parser, $name)
{
$this->currentTag = "";

// set flag if exiting <channel> or <item> block
if ($name == "ITEM")
{
$this->count++;
$this->flag = 0;
}
else if ($name == "CHANNEL")
{
$this->flag = 0;
}
}

// character data handler
function characterData($parser, $data)
{
$data = trim(htmlspecialchars($data));
if ($this->currentTag == "TITLE" || $this->currentTag ==
"LINK" || $this->currentTag == "DESCRIPTION")
{
// add data to $channels[] or $items[] array
if ($this->flag == 1)
{

$this->items[$this->count][strtolower($this->currentTag)] .=
$data;
}
else if ($this->flag == 2)
{

$this->channel[strtolower($this->currentTag)] .= $data;
}
}
}

// return an associative array containing channel information
// (the $channel[] array)
```

```
function getChannelInfo()
{
return $this->channel;
}

// return an associative array of arrays containing item
information
// (the $items[] array)
function getItems()
{
return $this->items;
}

}
?>
```

If you're familiar with PHP classes, this should be fairly simple to understand. If you're not, flip forward to the end of this article for a link to a great article on how classes work, and then read the listing above again.

Before we proceed to use this class, I'm going to take a few minutes out to point out one line from it − the call to xml_set_object() above.

Now, how about using this class to actually generate a Web page with multiple content feeds?

```
<?
include("class.RDFParser.php");
// how many items to display in each channel
$maxItems = 5;
?>
<html>
<head>
<basefont face="Verdana">
<body>

<table width="100%" border="0" cellspacing="5"
cellpadding="5"> <tr>
<!-- first cell -->
<td valign=top align=left>
<font size="-1">
<?
// get and parse freshmeat.net channel
$f = new RDFParser();
$f->setResource("http://www.freshmeat.net/backend/fm-releases.rdf");
$f->parseResource();
$f_channel = $f->getChannelInfo();
$f_items = $f->getItems();
// now format and print it...
```

```
?>
The latest from <a href=<? echo $f_channel["link"]; ?>><? echo
$f_channel["title"]; ?></a> <br> <ul> <? // iterate through
items array
for ($x=0; $x<$maxItems; $x++) {
if (is_array($f_items[$x]))
{
// print data
$item = $f_items[$x];
echo "<li><a href=" . $item["link"] . ">" .
$item["title"] . "</a>";
}
}
?>
</ul>
</font>
</td>

<!-- second cell -->
<td align=center width=50%>
<i>Primary page content here</i>
</td>

<!-- third cell -->
<td valign=top align=left>
<font size="-1">
<?
// get and parse slashdot.org channel
$s = new RDFParser();
$s->setResource("http://slashdot.org/slashdot.rdf");
$s->parseResource();
$s_channel = $s->getChannelInfo();
$s_items = $s->getItems();
// now format and print it...
?>
The latest from <a href=<? echo $s_channel["link"]; ?>><? echo
$s_channel["title"]; ?></a> <br> <ul> <? // iterate through
items array
for ($x=0; $x<$maxItems; $x++) {
if (is_array($s_items[$x]))
{
// print data
$item = $s_items[$x];
echo "<li><a href=" . $item["link"] . ">" .
$item["title"] . "</a>";
}
}
?>
</ul>
```

```
</font>
</td>

</tr>
</table>

</body>
</head>
</html>
```

This is fairly simple. Once an instance of the class has been instantiated with the "new" keyword,

```
$f = new RDFParser();
```

class methods can be used to set the location of the RDF file to be parsed

```
$f->setResource("http://www.freshmeat.net/backend/fm-releases.rdf");
$f->parseResource();
```

and to retrieve the $channel and $items arrays for further processing.

```
<?
$f_channel = $f->getChannelInfo();
$f_items = $f->getItems();
?>

The latest from <a href=<? echo $f_channel["link"]; ?>><? echo
$f_channel["title"]; ?></a> <br> <ul> <? // iterate through
items array
for ($x=0; $x<$maxItems; $x++) {
if (is_array($f_items[$x]))
{
// print data
$item = $f_items[$x];
echo "<li><a href=" . $item["link"] . ">" .
$item["title"] . "</a>";
}
}
?>
</ul>
```

Each time you reload the script above, the appropriate RDF files will be fetched from the specified locations, parsed and displayed in the format required.

If you have a high−traffic site, you might find this constant "pull" to be more of a nuisance than anything else, especially if the RDF data in question is not updated all that often. In that case, it might be wise to explore the option of caching the RDF data locally, either by extending the example above to include caching features, or by using a cron job to download a local copy of the latest RDF file to your Web server every couple hours, and using the local copy instead of the "live" one.

A Free Lunch

Now, the class I've written above is meant primarily for illustrative purposes, and perhaps for low−traffic sites. If you're looking for something more professional, there are a number of open−source RDF parsers out there which bring additional features (including caching) to the table. Let's look at some examples of how they can be used.

The first of these is the RDF parser class, developed by Stefan Saasen for fase4 and available for free download from http://www.fase4.com/rdf/. This is a very full−featured RDF parser, and includes support for caching and authentication via proxy. Here's an example of how it might be used:

```
<html>
<head>
<style type="text/css">
body {font−family: Verdana; font−size: 11px;}
.fase4_rdf {font−size: 13px; font−family: Verdana}
.fase4_rdf_title
{font−size: 13px; font−weight : bolder;}
</style>
</head>
<body>
<?
// include class
include("rdf.class.php");

// instantiate object
$rdf = new fase4_rdf;

// set number of items to display
$rdf−>set_max_item(5);

// set RDF engine options
$rdf−>use_dynamic_display(true);
$rdf−>set_Options( array("image"=>"hidden",
"textinput"=>"hidden") );

// parse and display data
$rdf−>parse_RDF("http://www.freshmeat.net/backend/fm−releases.rdf");
$rdf−>finish();
?>
</body>
</html>
```

An option to this is the PHP RDF parser class developed by Jason Williams, available at http://www.nerdzine.net/php_rdf/ . This is a bare−bones PHP class, which exposes a few basic methods, but a large number of properties that you can manipulate to format the processed data to your satisfaction.

```
<html>
<head>
<basefont face="Verdana">
</head>
<body link="Red" vlink="Red" alink="Red">
<?
include("rdf_class.php");

// this needs to be a local file
$f = new rdfFile("./fm-releases.rdf");
$f->parse(True);
$f->ReturnTable(True, "black", "white", "100%");
?>
</body>
</html>
```

Documentation for both these classes is available on their respective Web sites.

Adding A Little Style

In case you don't like the thought of iterating through all those PHP arrays and marking them up with HTML, you also have the option to format and display the data using an XSLT stylesheet. PHP 4.1 comes with support for the Sablotron XSLT processor via a new XSLT API, which can be used to combine an XSLT stylesheet with an XML (or, in this case, RDF) document to easily transform XML markup into browser−readable HTML.

I'm not going to get into the details of this − take a look at the PHP manual, or at the links at the end of the article, for detailed information − but I will demonstrate what I mean with a simple example. First, here's the stylesheet:

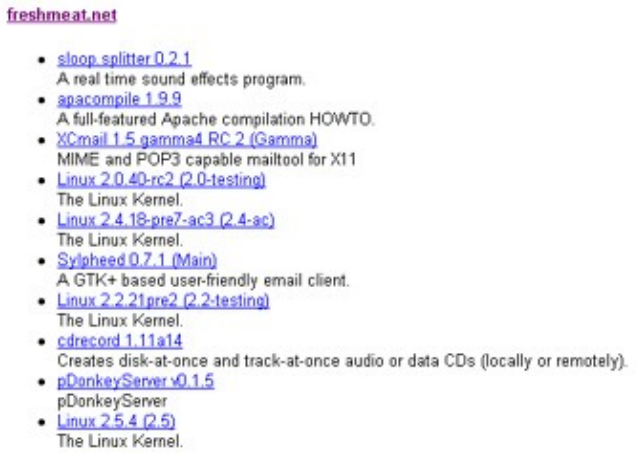
```
<?xml version="1.0"?>

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xmlns:rss="http://purl.org/rss/1.0/"
xmlns:dc="http://purl.org/dc/elements/1.1/" version="1.0">

<!−− main page −−>
<xsl:template match="/rdf:RDF">
<html>
<head>
<basefont face="Arial" size="2"/>
</head>
<body>
<xsl:apply−templates select="rss:channel" />
<ul>
<xsl:apply−templates select="rss:item" />
</ul>
</body>
</html>
</xsl:template>

<!−− channel −−>
<xsl:template match="rss:channel">
<b>
<a>
<xsl:attribute name="href"><xsl:value−of select="rss:link"
/></xsl:attribute>
<xsl:value−of select="rss:title" />
</a>
</b>
</xsl:template>

<!−− item −−>
```

```
<xsl:template match="rss:item">
<li />
<a>
<xsl:attribute name="href"><xsl:value-of select="rss:link"
/></xsl:attribute>
<xsl:value-of select="rss:title" />
</a>
<br />
<xsl:value-of select="rss:description" />
</xsl:template>

</xsl:stylesheet>
```

And here's the PHP script that combines the stylesheet above with the Freshmeat RDF document described previously to generate an HTML page:

```
<?php
// XML file
// this needs to be a local file
$xml = "fm-releases.rdf";

// XSLT file
$xslt = "fm.xsl";

// create a new XSLT processor
$xp = xslt_create();

// transform the XML file as per the XSLT stylesheet
// return the result to $result
$result = xslt_process($xp, $xml, $xslt);
if ($result)
{
// print it
echo $result;
}

// clean up
xslt_free($xp);

?>
```

Fairly simple and self-explanatory again, I think. The two documents are merged to produce the following composite output:



freshmeat.net

- sloop splitter 0.2.1
  A real time sound effects program.
- apacompile 1.9.9
  A full-featured Apache compilation HOWTO.
- XCmail 1.5.gamma4.RC.2 (Gamma)
  MIME and POP3 capable mailtool for X11
- Linux 2.0.40-rc2 (2.0-testing)
  The Linux Kernel.
- Linux 2.4.18-pre7-ac3 (2.4-ac)
  The Linux Kernel.
- Sylpheed 0.7.1 (Main)
  A GTK+ based user-friendly email client.
- Linux 2.2.21pre2 (2.2-testing)
  The Linux Kernel.
- cdrecord 1.11a14
  Creates disk-at-once and track-at-once audio or data CDs (locally or remotely).
- pDonkeyServer v0.1.5
  pDonkeyServer
- Linux 2.5.4 (2.5)
  The Linux Kernel.

This is an alternative, and perhaps simpler (though not all that optimal), method of turning RDF data browser−readable HTML. Note, however, that you will need to have an external program running (probably via cron) to update your local copy of the RDF file on a regular basis, since the PHP XSLT processor may have trouble accessing a remote file.

Homework

And that's about it for the moment. If you're interested in learning more about the technologies and techniques discussed in this article, you might want to consider visiting the following links:

The RSS 1.0 specification, at http://www.purl.org/rss/1.0/

An RSS timeline, at http://backend.userland.com/stories/rss091

The W3C's Web site on RDF, at http://www.w3.org/RDF/

A discussion of SAX and DOM programming in PHP, at
http://www.devshed.com/Server_Side/XML/XMLwithPHP

A discussion of XSLT transformation with PHP, at http://www.devshed.com/Server_Side/XML/XSLTrans

A discussion of PHP classes, at http://www.devshed.com/Server_Side/PHP/BackToClass

A discussion of XML basics, at http://www.devshed.com/Server_Side/XML/XMLBasic

A discussion of XSLT basics, at http://www.devshed.com/Server_Side/XML/XSLBasics

The PHP manual page for SAX functions, at http://www.php.net/manual/en/ref.xml.php

The PHP manual page for XSLT functions, at http://www.php.net/manual/en/ref.xslt.php

Till next time...stay healthy! Note: All examples in this article have been tested on Linux/i386 with Apache 1.3.12 and PHP 4.1.1. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!