

Session 10 – April 25, 2005

Outline

Fetching network resources via http
 with fopen()
 with the cURL extension
 with PEAR HTTP/Request
 with native network socket functions
Caching with PEAR Cache_Lite
Parsing XML: Adding RSS feeds to a site
 In PHP 4 with XML parser functions
 In PHP 5 with SimpleXML

Fetching network resources via http

Sometimes your site might need to grab a resource off some other site. If it's an image that you need, one solution is dead simple: use an html img tag with the fully qualified URL as its src attribute. But suppose you or your client decides you want to add something like Yahoo RSS feeds to your site and you need to grab an XML document. There are at least four possible solutions, and one of them – using a socket -- is guaranteed to be available with any PHP installation.

Option 1: fopen()

We met `fopen()` a few weeks ago while learning about file I/O. When the configuration directive `allow_url_fopen` is turned on, `fopen()` can open remote files via ftp and http as well as local files. In order to fetch the latest weird news story headlines from `rss.news.yahoo.com`, you would simply

```
$fp = fopen('http://rss.news.yahoo.com/rss/oddlyenough', 'r')  
    or die ("Error fetching RSS data");
```

to get a file pointer that you could then read through chunk by chunk with something like

```
while ($data = fread($fp, 256)) {  
    // do something with $data  
}
```

Or if you're confident that the remote file is not so large that memory consumption is an issue, simply

```
$data = file_get_contents('http://rss.news.yahoo.com/rss/oddlyenough');
```

The potential catch is that if `allow_url_fopen` is turned off and you cannot override it locally via `ini_set()` or a `.htaccess` directive, which is supposed to be the case in PHP $\geq 4.3.8$, then you are out of luck and must try something else.

Option 2: the cURL extension

cURL (<http://curl.haxx.se/>) is a library created by Daniel Stenberg that allows you to communicate with servers via `http`, `https`, `ftp`, `gopher`, `telnet`, `dict`, `file`, and `ldap` protocols. `libcurl` also supports HTTPS certificates, HTTP POST, HTTP PUT, FTP uploading (this can also be done with PHP's `ftp` extension), HTTP form based upload, proxies, cookies, and user+password authentication. cURL it is fairly ubiquitous, so it is likely to be installed on shared hosting systems, and if you control your own PHP installation, you can install cURL and build PHP with cURL support enabled (or load the DLL on Windows systems).

cURL is fast – reportedly faster than using native PHP socket functions -- flexible, reliable, and reasonably friendly. Here is how you would get that RSS feed into a string called `$data` using cURL:

```
$ch = curl_init("http://rss.news.yahoo.com/rss/oddlyenough");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$xml = curl_exec($ch);
curl_close($ch);
```

`curl_init()` takes a URL as argument and returns a *curl handle* (`$ch` in our example), which you use from that point on to set options and execute commands. By default, `curl_exec()` outputs the remote server response directly to the standard output (e.g., the browser) so if you want to capture the output in a string (e.g., for further examination or processing) you need to set the `CURLOPT_RETURNTRANSFER` to `true`. The moment of truth is `curl_exec()`; then you should close the connection with `curl_close()`. The functions and options are fairly numerous. See <http://php.net/manual/en/ref.curl.php> for complete documentation and some code examples.

Option 3: PEAR HTTP_Request

If you are fond of PEAR then another choice is the PEAR `HTTP_Request` class.

```
require('HTTP/Request.php');
$request = new HTTP_Request('http://rss.news.yahoo.com/rss/oddlyenough');
!PEAR::isError($request->sendRequest()) or die("error fetching RSS data");
```

```
$xml = $request->getResponseBody();
```

Here, all we had to do was load the class, instantiate an `HTTP_Request` object, and call two methods on it: `sendRequest()` followed by `getResponseBody()`. The `sendRequest()` method will store the response body in the `HTTP_Request` object; it will likewise fetch and save the response headers, which then become available through the `getResponseHeader()` method. See <http://pear.php.net/manual/en/package.http.http-request.php> for more information.

Option 4: Open a socket and speak HTTP yourself

If for some reason you cannot or prefer not to use the above PEAR package, you can't use `fopen()`, and you can't use `cURL`, all is not lost. Any PHP installation – even PHP 3.0.7 – supports network socket functions. Sockets are a comparatively low-level method for communication between two programs running on different machines across a network. A socket is defined as "the endpoint in a connection." Sockets can also be used for communication between processes within the same computer.

The negative of using sockets for this task is that it's more coding work than a higher-level API such as `cURL` or PEAR `HTTP_Request`. On the up side the sockets extension is fun to play with. Indeed, if you examine the source code of `HTTP_Request` you will see that it relies on another PEAR class called `Net_Socket`; and if you in turn look in that, you will see PHP socket functions in there. If performance is an important consideration, you might not like the idea of loading all that PEAR code just to fetch a little xml.

In earlier sessions we have discussed the HTTP protocol, its methods GET and POST, http headers and the request/response paradigm. When you do socket communications, you open a socket to the remote server and port, and speak the appropriate protocol yourself – in this case, HTTP – rather than having an API do it for you.

Here is a function that fetches an RSS file and returns it as a string, without relying on anything except plain old PHP:

```
function fetchRss($host,$path_to_resource) {  
  
    $sock = fsockopen($host,80,$errno,$errstr,15) ;  
    if (! $sock) {  
        trigger_error($errstr,E_USER_WARNING);  
        return false;  
    }  
  
    fwrite($sock,  
        "GET $path_to_resource HTTP/1.1\r\nHost: $host\r\nConnection:  
close\r\n\r\n");  
    // set a flag that tells us whether we're past the HTTP headers  
    $isBody = false;  
    // put the body in here  
    $body = '';  
    while (!feof($sock)) {  
  
        $data = fgets($sock, 128);
```

```

    // if we see the opening xml tag, we are in the
    // body of the server response
    if (strstr($data,'<?xml')) {
        $isBody = true;
        $body .= $data;
        continue;
    }
    if ($isBody) {
        $body .= $data;
        // if we see the closing rss tag,
        // we do not want any more $data
        if (stristr($data,'</rss>')) {
            break;
        }
    }
}
fclose($sock);
return $body;
}

```

Our function takes two arguments: a host name, and a path to the resource in question.

```
$sock = fsockopen($host, 80, $errno, $errstr, 15)
```

The `fsockopen()` function can take up to five arguments: host, port, two variable names in which to store any error number and an error message in that order, and a timeout (in seconds) after which to give up. If it succeeds it returns a socket handle that you can use much the way you would a file pointer: you read from and write to it with the same file I/O functions.

We open a socket and manually send an HTTP GET request. Then we read the response through same socket.

Most of the work here is due to the fact that we have to separate the response header from the body by hand. Before the `while()` loop, we initialize to false a flag that indicates whether or not we have reached the part of the document that contains the xml prologue – meaning the HTTP headers are over. Once we do, we set the flag to true and start appended incoming data to the string `$body`. Once we see the closing `</rss>` tag, we think we've reached the end of the xml data that interests us so we exit the while loop by saying `break` -- although this should not be necessary if the document is well-formed.

Note that if the web server on the other end were to give us a 404 response, that would not be a socket communications error; it would be an error at the http response level. The xml prolog would most likely never come, and the return value `$body` would be an empty string -- which evaluates to false. Thus the caller can check the return value and find that the resource was not returned, although there would be no way to figure out why. The error handling could be further refined but this will do for demo purposes (-:

When we call our function like so, or if we run any of the other above snippets that fetch this xml document via `fopen`, `cURL` or `PEAR HTTP_Request`

```
$data = fetchRss("rss.news.yahoo.com", "/rss/oddlyenough");
```

then the string \$data ends up containing something like this (as of December 2, 2004 at around 4:00 pm Eastern time):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<rss version="2.0">
<channel>
<title>Yahoo! News: Oddly Enough</title>
<copyright>Copyright (c) 2004 Yahoo! Inc. All rights reserved.</copyright>
<link>http://news.yahoo.com/news?tmpl=index&cid=757</link>
<description>Oddly Enough</description>
<language>en-us</language>
<lastBuildDate>Wed, 01 Dec 2004 20:51:45 GMT</lastBuildDate>
<ttl>5</ttl>
<image>
<title>Yahoo! News</title>
<width>142</width>
<height>18</height>
<link>http://news.yahoo.com/</link>
<url>http://us.il.yimg.com/us.yimg.com/i/us/nws/th/main_142.gif</url>
</image>
<item>
<title>Lava Lamp Left on Stove Explodes, Kills Man (Reuters)</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.ya
hoo.com/news?tmpl=story2&u=/nm/20041201/od_nm/odd_lavalamp_dc</link>
<guid isPermaLink="false">nm/20041201/odd_lavalamp_dc</guid>
<pubDate>Wed, 01 Dec 2004 14:12:50 GMT</pubDate>
<description>Reuters - A Washington state man who placed a
lava lamp on a hot stove died when the lamp exploded and a
glass shard pierced his heart, police said on Tuesday.</description>
</item>
<item>
<title>Man Walks Free After Wife Forgives Murder Bid (Reuters)</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.ya
hoo.com/news?tmpl=story2&u=/nm/20041201/od_nm/crime_france_pardon_dc</lin
k>
<guid isPermaLink="false">nm/20041201/crime_france_pardon_dc</guid>
<pubDate>Wed, 01 Dec 2004 14:11:49 GMT</pubDate>
<description>Reuters - A man who blinded his wife in a
failed murder and suicide attempt won a suspended five-year
sentence on Tuesday after his spouse publicly forgave him and
begged the court to acquit him.</description>
</item>
<item>
<title>Flowering Phone Is Environmental Wake-Up Call (Reuters)</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.ya
hoo.com/news?tmpl=story2&u=/nm/20041201/od_nm/odd_phone_dc</link>
<guid isPermaLink="false">nm/20041201/odd_phone_dc</guid>
<pubDate>Wed, 01 Dec 2004 14:13:25 GMT</pubDate>
<description>Reuters - A rose is a phone? British
scientists seeking to protect the environment have designed a
biodegradable mobile phone cover that breaks down in soil when
discarded and sprouts a flower from a seed embedded inside the
case.</description>
</item>
<item>
```

```

<title>Urinal Named As Most Influencial Art (AP)</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.ya
hoo.com/news?tmpl=story2&amp;u=/ap/20041201/ap_on_fe_st/urinal_art</link>
<guid isPermaLink="false">ap/20041201/urinal_art</guid>
<pubDate>Wed, 01 Dec 2004 19:20:44 GMT</pubDate>
<description>AP - A porcelain urinal is the most influential work of modern
art, according to a survey released Wednesday.</description>
</item>
<item>
<title>Germans think they're well-hung - but they're not (Reuters)
</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.ya
hoo.com/news?tmpl=story2&amp;u=/nm/20041201/od_uk_nm/oukoe_germany_condoms</l
ink>
<guid isPermaLink="false">nm/20041201/oukoe_germany_condoms</guid>
<pubDate>Wed, 01 Dec 2004 16:58:34 GMT</pubDate>
<description>Reuters - Most German men wear condoms of the wrong size, a
condom distributor has said, after asking more
than 2,500 men to measure their erect penis.</description>
</item>
<item>
<title>Men Arrested for Dumping Dirt in a Forest (AP)</title>
<link>http://us.rd.yahoo.com/dailynews/rss/oddlyenough/*http://story.news.yah
oo.com/news?tmpl=story2&amp;u=/ap/20041201/ap_on_fe_st/dirt_dumping_arrest</l
ink>
<guid isPermaLink="false">ap/20041201/dirt_dumping_arrest</guid>
<pubDate>Wed, 01 Dec 2004 14:33:08 GMT</pubDate>
<description>AP - Two men have been arrested for dumping dirt in a national
forest. The Kootenai County Sheriff's Department said the men, who have
not been publicly identified, were arrested at a garage in Coeur d'Alene
where the dirt had been removed and the base apparently prepared for
paving.</description>
</item>
<!-- item elements deleted for brevity -->
</channel>
</rss>

```

Caching Data

You can see that writing code to fetch a remote resource is easy. But it would be bad form, if not downright abusive, to hit someone's else's site every single time someone requests your page, especially if you should have the good fortune to have a popular site. Further, the network access is an expensive operation that will hurt performance on your site if used indiscriminately.

The solution is to cache the RSS file and only update it every so often. A good way to implement caching is with the PEAR Cache_Lite package.

The Cache_Lite package consists of three main classes: the base class Cache_Lite itself, which is concerned with saving and retrieving disk files containing cached data; Cache_Lite_Output, which makes use of PHP's output buffering so that you can have your cake and eat it to by sending output to the browser and also caching it; and Cache_Lite_Function, which caches the output of a function call.

The *lite* in Cache_Lite should be construed not as lite on features, but as the opposite of fat and slow. It is a solid but easy to use library that can provide a substantial performance boost when

used judiciously. Unlike most PEAR classes, it does not extend the base class PEAR and only instantiates PEAR::Error when it has to. That means a significant up front savings in overhead as compared to a lot of PEAR packages.

Caching isn't just for saving local copies of remote data; it's suitable for any data that whose generation from scratch is expensive enough to warrant it. For example, if you use a database query to populate a select menu with data that changes a lot less frequently than it is read, you would probably benefit from caching those results instead of hitting the database again on every page request.

Caching a file with Cache_Lite

The first steps are to load the class and instantiate a Cache_Lite object. The constructor can take numerous options for fine-tuning speed versus safety, which you are encouraged to investigate further at <http://pear.php.net/manual/en/package.caching.cache-lite.cache-lite.cache-lite.php>. Perhaps the most critical options are the location of the cache directory and the lifetime (in seconds) that you want your cache to live; for the rest you can rely on the reasonable defaults. Remember that the server has to be able to read and write to the directory where you're storing cached data.

```
require('Cache/Lite.php');

// give this cache file its own id
$id = 'yahoo.oddlyenough';

// set an array of options which we will pass to the constructor
$options = array(
    'cacheDir' => './cache/',
    'lifeTime' => 3600      // 1 hour
);

$cache = new Cache_Lite($options);

// see if the data is cached
if ($data = $cache->get($id)) {

    echo "using cached data...<br />";
    echo $data;
// otherwise, generate the data and cache it for next time

} else {

    echo "no cache, fetching from network...<br />";
    include('./fetch_xml_with_socket.php');
    $data = fetchRss("rss.news.yahoo.com", "/rss/oddlyenough");
    echo $data;
    if (!$cache->save($data)) {
        die( "damn! could not write to cache" );
    }
}
```

It's important that your code not do anything that it might not have to do until it checks the cache

for a cached copy of the data; otherwise you are defeating the purpose of caching.

Caching buffered output with Cache_Lite_Output

Perhaps even more elegant is using `Cache_Lite_Output`. When you call its `start()` method, it will look for the data in the cache and output it automatically for you; otherwise, it's up to you to generate it -- meanwhile your `Cache_Lite_Output` object has automatically started buffering output. When you're through generating the output, you need only call the `end()` method to save the buffered output in the cache and flush it in one shot. Pretty nice.

```
$options = array(
    'cacheDir' => './cache/',
    'lifeTime' => 3600      // 1 hour
);

$cache = new Cache_Lite_Output($options);

if (!($cache->start('yahoo.oddlyenough'))) {

    include('./fetch_xml_with_socket.php');
    $data = fetchRss("rss.news.yahoo.com", "/rss/oddlyenough");
    echo '<pre>' . htmlspecialchars($data) . '</pre>';
    $cache->end(); // the buffered output is now stored in a cache file
    echo ".....used fresh data..... ";

} else {
    echo ".... and I got it from the cache, how cool is that? (-: " ;
}
```

Caching output with Cache_Lite_Function

Even more terse is the `Cache_Lite_Function` class.

```
require('./fetch_xml_with_socket.php');
require_once('Cache/Lite/Function.php');
$options = array(
    'cacheDir' => './cache/',
    'lifeTime' => 3600,
);

$cache = new Cache_Lite_Function($options);
$data = $cache->call('fetchRss', "rss.news.yahoo.com", "/rss/oddlyenough");
echo '<pre>', htmlspecialchars($data), '</pre>';
```

This technique requires you to load the function definition regardless of whether you're really going to need it, because the `call()` method requires a valid callback. Nevertheless, if the data that the function returns is cached, the cached data will be used. Otherwise the function is run *and* the returned data is cached automatically.

If you need to purge your cache now rather waiting for it to expire later, `Cache_Lite` provides `remove()` and `clean()` methods for removing specific items and for clearing the whole thing, respectively. See <http://pear.php.net/manual/en/package.caching.cache-lite.intro.php> for complete details.

The full drama of `Cache_Lite` will become more evident when we ask it to save us more work – both network access and parsing XML.

Event-based (SAX) XML in PHP 4 (or 5)

This discussion assumes only minimal familiarity with XML basics.

Thus far we have looked at ways to fetch an RSS feed from a remote server and cache the result. Now let's see how you can turn that XML data into HTML for presentation on your site. RSS (Really Simple Syndication, according to some) is a popular format for passing around digests of the latest news from a particular site; it enables web sites to share content with each other and keep themselves looking up-to-date. You saw an example of raw RSS a few pages ago.

There are two basic ways of handling XML input – and this is true of programming languages generally, not just PHP.

The first is *event-based* or SAX (Simple API for XML) style parsing in which you loop through the data and callback functions are fired when you hit certain elements. This has the advantage that it's fast, and can be fairly simple.

The second is DOM, or Document Object Model, where an entire XML document is turned into an in-memory, tree-like data structure. This takes up more memory, but offers that advantage of greater control and flexibility: you can navigate around and manipulate the various *nodes* in ways that are difficult or impossible with event-based parsing. DOM is also best for situations where you don't know for certain what data you are expecting and need to discover it programmatically. For something like an RSS feed, you can reasonably expect the data in a particular format.

In PHP 4, the DOM extension was experimental. In PHP 5, the revamped DOM extension is built in, and vastly improved. For brevity's sake we will forgo DOM in this discussion; see <http://us2.php.net/manual/en/ref.dom.php> for more information.

In both PHP 4 and 5, SAX style parsing is available, and code that works under PHP 4 works nicely under PHP 5 although the internals have changed.

Parsing XML weather data with SAX

This example uses the PEAR `XML_Parser` class to turn XML from National Weather Service into an HTML table. The source XML looks like this:

```

<current_observation>
<credit>NOAA's National Weather Service</credit>
<credit_URL>http://weather.gov/</credit_URL>
<image>
<url>http://weather.gov/images/xml_logo.gif</url>
<title>NOAA's National Weather Service</title>
<link>http://weather.gov</link>
</image>
<suggested_pickup>15 minutes after the hour</suggested_pickup>
<suggested_pickup_period>60</suggested_pickup_period>
<location>Newark International Airport, NJ</location>
<station_id>KEWR</station_id>
<latitude>40.40.57N</latitude>
<longitude>074.10.10W</longitude>
<elevation>NA</elevation>
<observation_time>Last Updated on Apr 24, 6:51 pm EDT</observation_time>
<weather>Mostly Cloudy</weather>
<temperature_string>52 F (11 C)</temperature_string>
<temp_f>52</temp_f>
<temp_c>11</temp_c>
<relative_humidity>43</relative_humidity>
<wind_string>From the Southwest at 10 MPH</wind_string>
<wind_dir>Southwest</wind_dir>
<wind_degrees>230</wind_degrees>
<wind_mph>10.35</wind_mph>
<wind_gust_mph>0</wind_gust_mph>
<pressure_string>29.42" (996.2 mb)</pressure_string>
<pressure_mb>996.2</pressure_mb>
<pressure_in>29.42</pressure_in>
<dewpoint_string>30 F (-1 C)</dewpoint_string>
<dewpoint_f>30</dewpoint_f>
<dewpoint_c>-1</dewpoint_c>
<heat_index_string>Not Applicable</heat_index_string>
<heat_index_f>Not Applicable</heat_index_f>
<heat_index_c>Not Applicable</heat_index_c>
<windchill_string>48 F (9 C)</windchill_string>
<windchill_f>48</windchill_f>
<windchill_c>9</windchill_c>
<visibility>10.00 mi.</visibility>
<two_day_history_url>http://www.weather.gov/data/obhistory/KEWR.html</two_day_history_url>
<ob_url>http://www.nws.noaa.gov/data/METAR/KEWR.1.txt</ob_url>
<disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
<copyright_url>http://weather.gov/disclaimer.html</copyright_url>
<privacy_policy_url>http://weather.gov/notice.html</privacy_policy_url>
</current_observation>

```

We subclass XML_Parser and create callbacks that get fired when certain xml “events” happen as the parser goes through the XML from top to bottom. We are also using caching so that we don't have to do much of anything if the data is in the cache.

```

require('Cache/Lite/Output.php');
$cache = new Cache_Lite_Output( array(
'cacheDir' => './cache/', 'lifeTime' => 3600
));
$id = 'ewr_weather_data';

if (!$cache->start($id)) {
    // load parent class
    require('XML/Parser.php');

```

```

//extend the class
class Weather_Parser extends XML_Parser {

    // hold character data here
    private $cdata;

    // implement handlers for opening & closing tags,
    // and stuff in between
    function startHandler($xp,$element,&$attribs) {
        $this->cdata = '';
        if ($element=='CURRENT_OBSERVATION') {
            echo '<table border="1" cellpadding="2"
cellspacing="0">';
            return;
        }
        if ($element=='IMAGE') {return;}
        echo "<td style=\"text-align:right;font-
weight:bold\">$element&nbsp;</td>";
    }

    function cdataHandler($xp,$data) {
        $this->cdata .= $data;
    }

    function endHandler($xp,$element) {
        if ($element=='CURRENT_OBSERVATION') {echo '</table><br /
>';return;}

        if ($element=='IMAGE') {return;}
        print "<td>$this->cdata</td></tr>";
    }
}

$ch = curl_init("http://weather.gov/data/current_obs/KEWR.xml");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$xml = curl_exec($ch);
curl_close($ch);
$parser = new Weather_Parser();
// $fh = fopen('./weather_data.xml','r') or die('shit');
// $parser->setInputFile($fh)
// $parser->parse();
$parser->parseString($xml);
$cache->end();
} else {
    echo "<p>The whole thing came from the cache. Cool, n'est-ce pas?</p>";
}
}

```

The output of the above in a browser looks like this:

| | |
|--------------------------------|---|
| CREDIT | NOAA's National Weather Service |
| CREDIT_URL | http://weather.gov/ |
| URL | http://weather.gov/images/xml_logo.gif |
| TITLE | NOAA's National Weather Service |
| LINK | http://weather.gov |
| SUGGESTED_PICKUP | 15 minutes after the hour |
| SUGGESTED_PICKUP_PERIOD | 60 |
| LOCATION | Newark International Airport, NJ |
| STATION_ID | KEWR |
| LATITUDE | 40.40.57N |
| LONGITUDE | 074.10.10W |
| ELEVATION | NA |
| OBSERVATION_TIME | Last Updated on Apr 24, 9:51 pm EDT |
| WEATHER | Overcast |
| TEMPERATURE_STRING | 48 F (9 C) |
| TEMP_F | 48 |
| TEMP_C | 9 |
| RELATIVE_HUMIDITY | 61 |
| WIND_STRING | From the West at 8 MPH |
| WIND_DIR | West |
| WIND_DEGREES | 250 |
| WIND_MPH | 8.05 |
| WIND_GUST_MPH | 0 |
| PRESSURE_STRING | 29.47" (997.9 mb) |
| PRESSURE_MB | 997.9 |
| PRESSURE_IN | 29.47 |
| DEWPOINT_STRING | 35 F (2 C) |
| DEWPOINT_F | 35 |
| DEWPOINT_C | 2 |
| HEAT_INDEX_STRING | Not Applicable |
| HEAT_INDEX_F | Not Applicable |

and so on. This example simply skips elements that we know to have child nodes; that way we avoid printing the character data more than once. You can imagine the possibilities, e.g, just pick out data elements that interest you and putting together a little weather report for inclusion in a web site. It's more work than the DOM-based approach, but it's fast and less memory-intensive.

Parsing RSS/XML data with SAX in PHP 4 or 5

The following example is borrowed from <http://www.sitepoint.com/article/php-xml-parsing-rss->

1-0, but the techniques it uses are well known. If you are using PHP 4, you could do something like this. It is rather a lot of work compared to PHP5's SimpleXML, which we will look at next.

```
class RSSParser {

    var $insideitem = false;
    var $tag = "";
    var $title = "";
    var $description = "";
    var $link = "";

    function startElement($parser, $tagName, $attrs) {

        if ($this->insideitem) {
            $this->tag = $tagName;
        } elseif ($tagName == "ITEM") {
            $this->insideitem = true;
        }
    }

    function endElement($parser, $tagName) {

        if ($tagName == "ITEM") {
            printf("<p><b><a href='%s'>%s</a></b><br />",
                trim($this->link),htmlspecialchars(trim($this->title)));
            printf("%s</p>",
                htmlspecialchars(trim($this->description)));
            $this->title = "";
            $this->description = "";
            $this->link = "";
            $this->insideitem = false;
        }
    }

    function characterData($parser, $data) {

        if ($this->insideitem) {
            switch ($this->tag) {
                case "TITLE":
                    $this->title .= $data;
                    break;
                case "DESCRIPTION":
                    $this->description .= $data;
                    break;
                case "LINK":
                    $this->link .= $data;
                    break;
            }
        }
    }
}

// create parser
$xml_parser = xml_parser_create();

// instantiate your class
$rss_parser = new RSSParser();

// inform the parser what object you are using
```

```

xml_set_object($xml_parser,$rss_parser);

// set your callback methods
xml_set_element_handler($xml_parser, "startElement", "endElement");
xml_set_character_data_handler($xml_parser, "characterData");

// fetch xml
$fp = fopen("http://rss.news.yahoo.com/rss/oddlyenough",'r')
    or die("Error reading RSS data.");

// chunk your way through it parsing data
while ($data = fread($fp, 4096))
    xml_parse($xml_parser, $data, feof($fp))
        or die(sprintf("XML error: %s at line %d",
            xml_error_string(xml_get_error_code($xml_parser)),
            xml_get_current_line_number($xml_parser)));
fclose($fp);
xml_parser_free($xml_parser);

```

The procedure is to set callback functions that get invoked when certain events happen. There are several different events for which you can assign callbacks – see <http://www.php.net/manual/en/ref.xml.php> for details – but the most important events are usually `xml_set_element_handler()` and `xml_set_character_data_handler()`. The former expects three arguments: the parser, and two callbacks, one for the opening element tags and one for the closing tags of elements. These two functions are invoked whenever the parser encounters the opening and closing tags (respectively) of any element.

Since you never know whether the current chunk of data contains an opening or a closing or both, this program uses a flag to keep track of whether we are currently inside an element or not -- the `endElement` handler turns it off, outputs the all html for the current item in the RSS feed, and resets the object's member variables to empty strings.

The `startElement` handler turns on the 'insideItem' flag and sets a member variable that keeps track of the name of the current element.

The `characterData` handler checks whether we are currently inside an element, and if so, appends the data to the appropriate member variable.

The logic is rather complicated but not horribly so, once you wrap your mind around it --and you can easily steal this code and change the URL to parse any RSS feed. Adapted to use caching, it would do just fine.

A much easier solution is to use the PEAR XML_RSS class:

<http://pear.php.net/manual/en/package.xml.xml-rss.php>. This would also be an perfectly serviceable solution, especially if optimized with caching. Here is an example straight from their web site:

```

<?php
require_once "XML/RSS.php";

$rss =& new XML_RSS("http://slashdot.org/slashdot.rdf");
$rss->parse();

```

```

echo "<h1>Headlines from <a href=\"http://slashdot.org\">Slashdot</a></h1>\n";
echo "<ul>\n";

foreach ($rss->getItems() as $item) {
    echo "<li><a href=\"\" . $item['link'] . \">\" . $item['title'] .
\"</a></li>\n";
}

echo "</ul>\n";
?>

```

Still, an even better solution, arguably, is to use PHP 5 and SimpleXML.

Parsing RSS with SimpleXML in PHP 5

SimpleXML is one of those rare instances where something that calls itself Simple actually is. Here's a program that accomplishes the same thing as the above but with vastly less effort on the programmer's part:

```

include('./fetch_xml_with_socket.php');

$rss = simplexml_load_string(fetchRss
("rss.news.yahoo.com", "/rss/oddlyenough"));

foreach ($rss->channel->item as $item) {

    printf('<p><strong><a href=\"%s\">%s</a></strong><br />%s</p>',
        $item->link,
        $item->title,
        $item->description);
}

```

SimpleXML can load a string of XML and create an object whose properties correspond to the elements in the XML document. Then you iterate through the nested `$item` objects printing the properties that interest you, and you're done. See <http://www.php.net/manual/en/ref.simplexml.php> for full details (where at this writing it still says the extension is experimental. That may be so in PHP 4, but not in 5, AFAIK).

Here's the same program again, with caching:

```

require('Cache/Lite/Output.php');
$id = 'yahoo.oddlyenough.simpleXML';

$options = array(
    'cacheDir' => './cache/',
    'lifeTime' => 3600 // 1 hour
);

$cache = new Cache_Lite_Output($options);

if (!($cache->start($id))) {

    include('./fetch_xml_with_socket.php');
    $rss = simplexml_load_string(fetchRss
("rss.news.yahoo.com", "/rss/oddlyenough"));

```

```

foreach ($rss->channel->item as $item) {

    printf('<p><strong><a href="%s">%s</a></strong><br />%s</p>',
        $item->link,
        $item->title,
        $item->description);

}

$cache->end();
echo ".....used fresh data..... ";

} else {

    echo ".... and I got it from the cache, how cool is that? (-: " ;

}

```

Now all the heavy lifting – fetching XML across the network and parsing it – is cached so that it only happens once per hour.

Finally, here is a slightly more involved version of the relevant part of the above program, but with a bit better error handling. It only caches and outputs data if everything works as planned.

```

if (!($cache->start($id))) {

    include('./fetch_xml_with_socket.php');

    if ($rss = @simplexml_load_string(
        @fetchRss("rss.news.yahoo.com", "/rss/oddyenough")))
    {
        foreach ($rss->channel->item as $item) {

            printf('<p><strong><a href="%s">%s</a></strong><br />%s</p>',
                $item->link,
                $item->title,
                $item->description);

        }

        // cache and flush buffer
        $cache->end();
        echo ".....used fresh data and cached it..... ";

    } else { // it failed for some reason

        echo "rss feed not available";

    }

} else {

    echo ".... and I got it from the cache, how cool is that? (-: " ;

}

```

Further reading: Where to Go From Here

Adam Trachtenberg, *Upgrading to PHP 5* (O'Reilly)

David Sklar, *Essential PHP Tools* (Apress)

Harry Fuecks, *The PHP Anthology* Vol I and/or II (Sitepoint)

last update 24-Apr-2005 11:29 pm