How do you know who is doing what and where on your site? Chances are you have an Apache-style log for your site, and you just need to learn how to mine it for valuable information. Learn about the format of Web server logs and how to access them in code. Along the way, apply a recipe to identify spider traffic from Web crawlers.

A user types in your site's URL into his or her browser. The user is directed from http://cool.si.te/ to http://cool.si.te/index.html, which loads the HTML, which causes CSS, JavaScript, and images to be loaded. Your Web server is keeping tabs on all this activity, even the little nuances that most users have no idea about. If anything goes wrong, from a pattern that looks like abuse to a missing image file, you already know how to find the traces in your log files. Maybe you also use Web traffic analyzers that read your log files to keep tabs on traffic trends on your site. But for the most part, Web server logs gather dust in the foot lockers of system administrators, which is a shame because there are a million neat uses for them. In this article I explore some ideas and techniques for getting extra value from Web server logs. I test the techniques against the popular Apache server, but since so many other tools use the same formats, you should find this information more broadly applicable.

Anatomy of Apache logs

Major errors in the configuration or operation of Apache are reported in an error log, but in this article I'll focus on access logs, which have an entry for each HTTP request. The classic format originated at the National Center for Supercomputing Applications (NCSA), home of key Web innovations such as Mosaic (which became the Netscape browser), HTTPd (which was the main code base for the first Apache release), and Common Gateway Interface (CGI), which was the first mechanism for dynamic Web content. NCSA HTTPd defaulted to what's called Common log format, which was then adopted by Apache. Common log format is still used by many tools on the Web.

Common log format

Listing 1 is an example of a Common log format line:
Working with Web server logs

Listing 1. Common log format line

```
125.125.125.125 - uche [20/Jul/2008:12:30:45 +0700] "GET /index.html HTTP/1.1" 200
```

Table 1 explains the fields.

### Table 1. Fields in a common log format line

<table>
<thead>
<tr>
<th>Field name</th>
<th>Example value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>125.125.125</td>
<td>IP address or host name of the HTTP client that made the request</td>
</tr>
<tr>
<td>identd</td>
<td>-</td>
<td>Authentication Server Protocol (RFC 931) identifier for the client; this field is rarely used. If unused it's given as &quot;.&quot;</td>
</tr>
<tr>
<td>username</td>
<td>uche</td>
<td>HTTP authenticated user name (via 401 response handshake); this is the login and password dialog you see on some sites, as opposed to a login form embedded in a Web page, where your ID information is stored in a server-side session. If unused (for example, when the request is for an unrestricted resource) it's given as &quot;.&quot;</td>
</tr>
<tr>
<td>date/time</td>
<td>[20/Jul/2008:12:30:45 +0700]</td>
<td>Date then time then timezone, in the format [dd/MMM/yyyy:hh:mm:ss +-hhmm]</td>
</tr>
<tr>
<td>request line</td>
<td>&quot;GET /index.html HTTP/1.1&quot;</td>
<td>The leading line of the HTTP request, which includes the method (&quot;GET&quot;), the requested resource, and the HTTP protocol version</td>
</tr>
<tr>
<td>status code</td>
<td>200</td>
<td>Numeric code used in the response to indicating the disposition of the request, for example to indicate success, failure, redirect, or authentication requirement</td>
</tr>
<tr>
<td>bytes</td>
<td></td>
<td>Number of bytes transferred in the body of the response</td>
</tr>
</tbody>
</table>

 Combined log format

Many tools now default to a richer variation, combined log format. Listing 2 is an example of this.

Listing 2. Combined log format line

```
125.125.125.125 - uche [20/Jul/2008:12:30:45 +0700] "GET /index.html HTTP/1.1" 200
```

"http://www.ibm.com/" "Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9a8) Gecko/2007100619 GranParadiso/3.0a8" "USERID=Zepheira;IMPID=01234"

This should all be on one line, but I broke it into several to meet article formatting restrictions. The combined log format is the common format plus three additional fields—referrer, user agent, and cookie. You can omit the cookie field (quotes and all), or you can omit cookie and user agent, or you can omit all three. Table 2 has more on these added fields.

### Table 2. Additional fields in a combined log format line

<table>
<thead>
<tr>
<th>Field name</th>
<th>Example value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>referrer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>user agent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cookie</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When a user agent follows a link from one site to another, it often reports to the second site which URL referred it.

A string providing information about the user agent that made the request (for example, a browser version or a Web crawler).

The actual key/value pairs of any cookie that were sent by the HTTP server can send back to the client in the response.

Most people use their Web server’s default, but you can easily customize the format of Apache logs. Obviously, if you do so, you would have to adjust accordingly, including tweaking much of the code presented in this article.

Feeding log information to programs

You’ve seen how well structured these formats are. It’s fairly straightforward to use regular expressions to get at the information. Listing 3 is a simple program to parse a log line and write a summary of its information. It’s written in Python, but the meat of it is in a regular expression, so it is easily ported to any other language.

Listing 3. Code to parse a log line

```python
import re

COMBINED_LOGLINE_PAT = re.compile(  
    r'(?P<origin>\d+\.\d+\.\d+\.\d+) '  
    + r'(?P<identd>-|\w*) (?P<auth>-|\w*) '  
    + r'\[(?P<date>[^\[]\[:]+):(?P<time>\d+:\d+:\d+) (?P<tz>[\-+]?\d\d\d)\] '  
    + r'"(?P<method>\w+) (?P<path>\[\S\]+) (?P<protocol>[^\"]+)" (?P<status>\d+) '  
    + r'( (?P<referrer>"\[^\"]*"))( (?P<client>"\[^\"]*"))( (?P<cookie>"\[^\"]*"))?'?\s*Z'  
)

logline = raw_input("Paste the Apache log line then press enter: ")
match_info = COMBINED_LOGLINE_PAT.match(logline)
print match_info
print #Add a new line

#Print all named groups matched in the regular expression
for key, value in match_info.groupdict().items():
    print key, "", value
```

The pattern `COMBINED_LOGLINE_PAT` is designed for the combined form, but since the combined form is only three optional fields in addition to the common form, this pattern works fine for both. The pattern uses a Python-specific feature, named capturing groups, in order to assign a logical name to each field of the log line. To port to other regular expression flavors just use regular groups and refer to the fields by numerical order. Notice how fine grained I made the pattern. Rather than grabbing the status line as a unit, it grabs the HTTP method, the request path, and the protocol version separately for increased convenience. The date/time is also broken down into date, time, and timezone. Figure 1 shows the output from running Listing 3 against the sample log line in Listing 2.
Avoiding spiders

Many of the more interesting things you can learn from your log files require that you differentiate access by robots from access by humans. The major search engines such as Google and Yahoo! use very aggressive indexing, and if your site is even moderately popular, your logs might be dominated by traffic from these spiders. It’s nearly impossible to weed out spider traffic with 100% accuracy, but you can get most of the way by checking for common robot patterns in log files. The “client” field is key here. Listing 4 is another Python program, also designed for easy porting to other languages. It takes a log file piped into standard input, and it pipes back out all the lines except those it determines to be robot traffic.

Listing 4. Weed out search engine spider traffic from a log file

```python
import re
import sys

# This regular expression is the heart of the code.
# Python uses Perl regex, so it should be readily portable
# The r'' string form is just a convenience so you don't have to escape backslashes
COMBINED_LOGLINE_PAT = re.compile(  
    r'(\(?P<origin>\d+\.\d+\.\d+\.\d+) '  
    + r'(\(?P<identd>-|\w*) (?P<auth>-|\w*) '  
    + r'\[(?P<date>^[^\[\]:*]+):(?P<time>\d+:\d+\d+) (?P<tz>[\-\+\d]*)\] '  
    + r'"(?P<method>\w+) (?P<path>\S+) (?P<protocol>[^"]+)" (?P<status>\d+) '  
    (\(?P<bytes>-|\d+) '  
    + r'( (?P<referrer>"[^"]*")\(( (?P<client>"[^"]*")\(( (?P<cookie>"[^"]*"))\))?\s*Z '  
)

# Patterns in the client field for sniffing out bots
BOT_TRACES = [  
    (re.compile(r".*http://help\.yahoo\.com/help/us/ysearch/slurp\.*"),  
    "Yahoo robot"),  
    (re.compile(r".*http://www\.google\.com/bot\.html\.*"),  
    "Google robot"),  
    (re.compile(r".*http://about\.ask\.com/en/docs/about/webmasters\.shtml\.*"),  
    "Ask Jeeves/Teoma robot"),  
    (re.compile(r".*http://search\.msn\.com/\msnbot\.htm\.*"),  
    "MSN robot"),
```

Figure 1. Output from Listing 3
for line in sys.stdin:
    match_info = COMBINED_LOGLINE_PAT.match(line)
    if not match_info:
        sys.stderr.write("Unable to parse log line\n")
        continue
    isbot = False
    for pat, botname in BOT_TRACES:
        if pat.match(match_info.group('client')):
            isbot = True
            break
    if not isbot:
        sys.stdout.write(line)

The list of spider client regular expressions is not exhaustive. New search engines pop up all the time. You should be able to emulate the patterns in the listing to extend the list to include any new spiders you notice when reviewing traffic.

Basic tools for log stats

There are many popular tools for analyzing Web server logs and presenting statistics on the Web. Using the building blocks so far in this article, it's easy for you to develop your own specialized presentation of log information. One additional building block is conversion from Apache log format to JavaScript Object Notation (JSON). If you have the information in JSON, you can easily analyze it, manipulate it, and present it using JavaScript, including client side.

You might not even need to write the tools yourself. In this section I'll show you how to convert an Apache log file to JSON conversions used by Exhibit, the powerful data presentation tool from MIT's SIMILE project. I covered this tool in an earlier article, "Practical linked, open data with Exhibit" (see Related topics). All you have to do is give it JSON, and Exhibit can create a rich, dynamic system for displaying, filtering, and searching data. Listing 5 (apachelog2exhibit.py) builds on the earlier examples, but converts an Apache log to the Exhibit flavor of JSON.

Listing 5 (apachelog2exhibit.py). Convert non-spider traffic log entry items to Exhibit JSON

```python
import re
import sys
import time
import httplib
import datetime
import itertools
# You'll need to install the simplejson module
# http://pypi.python.org/pypi/simplejson
import simplejson

# This regular expression is the heart of the code.
# Python uses Perl regex, so it should be readily portable
# The r'' string form is just a convenience so you don't have to escape backslashes
COMBINED_LOGLINE_PAT = re.compile(
```
# Patterns in the client field for sniffing out bots

BOT_TRACES = [
    (re.compile(r".*http://help\.yahoo\.com/help/us/ysearch/slurp\."), "Yahoo robot"),
    (re.compile(r".*http://www\.google\.com/bot\.html\."), "Google robot"),
    (re.compile(r".*http://about\.ask\.com/en/docs/about/webmasters.shtml\."), "Ask Jeeves/Teoma robot"),
    (re.compile(r".*http://search\.msn\.com/msnbot\.htm\."), "MSN robot"),
    (re.compile(r".*http://www\.entireweb\.com/about/search_tech/speedy_spider\./\.*"), "Speedy Spider"),
    (re.compile(r".*http://www\.baidu\.com/search/spider_jp\./html\."), "Baidu spider"),
    (re.compile(r".*http://www\.gigablast\.com/spider\./html\."), "Gigabot robot"),
]

MAXRECORDS = 1000

# Apache's date/time format is very messy, so dealing with it is messy
# This class provides support for managing timezones in the Apache time field
# Reuses some code from: http://seehuhn.de/blog/52

class timezone(datetime.tzinfo):
    def __init__(self, name="+0000"):
        self.name = name
        seconds = int(name[:-2])*3600+int(name[-2:])*60
        self.offset = datetime.timedelta(seconds=seconds)

    def utcoffset(self, dt):
        return self.offset

    def dst(self, dt):
        return timedelta(0)

    def tzname(self, dt):
        return self.name

def parse_apache_date(date_str, tz_str):
    """
    Parse the timestamp from the Apache log file, and return a datetime object
    """
    tt = time.strptime(date_str, "%d/%b/%Y:%H:%M:%S")
    tt = tt[:6] + (0, timezone(tz_str))
    return datetime.datetime(*tt)

def bot_check(match_info):
    """
    Return True if the matched line looks like a robot
    """
    for pat, botname in BOT_TRACES:
        if pat.match(match_info.group('client')):
            return True
    break

    return False

entries = []

# enumerate lets you iterate over the lines in the file, maintaining a count variable
# itertools.islice lets you iterate over only a subset of the lines in the file
for count, line in enumerate(itertools.islice(sys.stdin, 0, MAXRECORDS)):
    match_info = COMBINED_LOGLINE_PAT.match(line)
    if not match_info:
        sys.stderr.write("Unable to parse log line\n")
        continue
    # If you want to include robot clients, comment out the next two lines
    if bot_check(match_info):
        continue
    entry = {}
    timestamp = parse_apache_date(match_info.group('ts'), match_info.group('tz'))
    timestamp_str = timestamp.isoformat()
    # To make Exhibit happy, set id and label fields that give some information
    # about the entry, but are unique across all entries (ensured by appending count)
    entry['id'] = match_info.group('origin') + ':' + timestamp_str + ':' + str(count)
    entry['label'] = entry['id']
    entry['origin'] = match_info.group('origin')
    entry['timestamp'] = timestamp_str
    entry['path'] = match_info.group('path')
    entry['method'] = match_info.group('method')
    entry['protocol'] = match_info.group('protocol')
    entry['status'] = match_info.group('status')
    entry['status'] += ' ' + httplib.responses[int(entry['status'])]
    if match_info.group('bytes') != '-':
        entry['bytes'] = match_info.group('bytes')
    if match_info.group('referrer') != '"-"':
        entry['referrer'] = match_info.group('referrer')
    entry['client'] = match_info.group('client')
    entries.append(entry)

print simplejson.dumps({'items': entries}, indent=4)

Just pipe in the Apache log files to `python apachelog2exhibit.py`, and capture the output JSON. Listing 6 is a brief example of output JSON.

Listing 6. Sample Exhibit JSON from Apache log

```
{
  "items": [
    {
      "origin": "208.111.154.16",
      "status": "200 OK",
      "protocol": "HTTP/1.1",
      "timestamp": "2009-04-27T08:21:42-05:00",
      "bytes": "2638",
      "auth": "-",
      "label": "208.111.154.16:2009-04-27T08:21:42-05:00:2",
      "identd": "-",
      "method": "GET",
      "referrer": "-",
      "path": "/uche.ogbuji.net",
      "id": "208.111.154.16:2009-04-27T08:21:42-05:00:2"
    },
    {
      "origin": "65.103.181.249",
      "status": "200 OK",
      "protocol": "HTTP/1.1",
      "timestamp": "2009-04-27T09:11:54-05:00",
      "bytes": "6767",
      "auth": "-",
      "label": "65.103.181.249:2009-04-27T09:11:54-05:00:4",
      "identd": "-",
      "method": "GET",
      "referrer": "-",
      "path": "/authenticate",
      "id": "65.103.181.249:2009-04-27T09:11:54-05:00:4"
    }
  ]
}
```

To use Exhibit you create an HTML page that loads the Exhibit library JavaScript as well as the JSON for the data. Listing 7 is a very simple Exhibit HTML page for the log file information.

**Listing 7. HTML for Exhibit log viewer**

```html
<html>
<head>
<title>Apache log entries</title>
<link href="logview.js" type="application/json" rel="exhibit/data" />
<script src="http://static.simile.mit.edu/exhibit/api-2.0/exhibit-api.js" type="text/javascript"></script>
<script src="http://static.simile.mit.edu/exhibit/extensions-2.0/time/time-extension.js" type="text/javascript"></script>
<style>
#main { width: 100%; }
#timeline { width: 100%; vertical-align: top; }
td { vertical-align: top; }
.entry { border: thin solid black; width: 100%; }
#facets { padding: 0.5em; width: 20%; }
.label { display: none; }
</style>
</head>
<body>
<h1>Apache log entries</h1>
<table id="main">
<tr>
<!-- The main display area for Exhibit -->
<td ex:role="viewPanel">
<div id="what-lens" ex:role="view"
ex:viewClass="Exhibit.TileView"
ex:label="What">
</div>
</td>
<!-- Timeline view for the feed data -->
<div id="timeline" ex:role="view"
ex:viewClass="Timeline"
ex:label="When"
ex:start=".timestamp"
ex:colorKey=".status"
ex:topBandUnit="day"
ex:topBandPixelsPerUnit="200"
ex:topBandUnit="week">
</div>
<!-- Boxes to allow users narrow down their view of feed data -->
<td id="facets">
</td>
</tr>
</table>
</body>
</html>
```
Figure 2 shows you one of the rich output views you get from just that simple HTML source. Exhibit does all the hard work for you. The facet boxes allow you to narrow down the items shown. For example, you might decide to analyze the access patterns for one origin address.

**Figure 2. Exhibit log file viewer ("What" view)**

Figure 3 shows you another one of the output views, the timeline view. To go to this view yourself, click **When** from the top of the default view.
Wrap up

There are many uses for information from log files. I've used it to decide when it's a good idea to apply some new Web technology, such as browser-side XSLT, by checking the proportion of the non-spider visitors to the sites that are using XSLT-capable browsers. I've used it to suggest useful tags for a Weblog entry, by reviewing referrer fields that come from search engines. There are many tools available to provide general statistics and analysis of log files, but there won't be an existing tool for every use of logs, so learning to process them directly is a valuable skill for a Web architect.
Related topics

- The open source **Apache HTTP Server** is the most widely used.
- Learn about the **log file formats** supported by the most recent version of Apache.
- Get introduced to this article's techniques in "**Using regular expressions,**" by David Mertz.
- Get a handle on the **many flavors of regular expressions**, their features, and corresponding syntax.
- PHP users can learn what they need to port the code in this article in "**Mastering regular expressions in PHP, Part 1**" and **Part 2**, by Martin Streicher.
- Learn more about Exhibit from "**Practical linked, open data with Exhibit,**" by Uche Ogbuji.
- Listing 5 uses the **simplejson** library.

© Copyright IBM Corporation 2009
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)