Log Files

In order to effectively manage a web server, it is necessary to get feedback about the activity and performance of the server as well as any problems that may be occurring. The Apache HTTP Server provides very comprehensive and flexible logging capabilities. This document describes how to configure its logging capabilities, and how to understand what the logs contain.

Overview

Related Modules	Related Directives
mod_log_config mod_log_forensic mod_logio mod_cgi	

The Apache HTTP Server provides a variety of different mechanisms for logging everything that happens on your server, from the initial request, through the URL mapping process, to the final resolution of the connection, including any errors that may have occurred in the process. In addition to this, third-party modules may provide logging capabilities, or inject entries into the existing log files, and applications such as CGI programs, or PHP scripts, or other handlers, may send messages to the server error log.

In this document we discuss the logging modules that are a standard part of the http server.

Security Warning

Anyone who can write to the directory where Apache httpd is writing a log file can almost certainly gain access to the uid that the server is started as, which is normally root. Do *NOT* give people write access to the directory the logs are stored in without being aware of the consequences; see the security tips document for details.

In addition, log files may contain information supplied directly by the client, without escaping. Therefore, it is possible for malicious clients to insert control-characters in the log files, so care must be taken in dealing with raw logs.

Error Log

Related Modules	Related Directives
core	ErrorLog ErrorLogFormat LogLevel

The server error log, whose name and location is set by the `ErrorLog` directive, is the most important log file. This is the place where Apache httpd will send diagnostic information and record any errors that it encounters in processing requests. It is the first place to look when a problem occurs with starting the server or with the operation of the server, since it will often contain details of what went wrong and how to fix it.

The error log is usually written to a file (typically `error_log` on Unix systems and `error.log` on Windows and OS/2). On Unix systems it is also possible to have the server send errors to `syslog` or pipe them to a program.

The format of the error log is defined by the ErrorLogFormat directive, with which you can customize what values are logged. A default is format defined if you don't specify one. A typical log message follows:

```
[Fri Sep 09 10:42:29.902022 2011] [core:error] [pid
35708:tid 4328636416] [client 72.15.99.187] File does not
exist: /usr/local/apache2/htdocs/favicon.ico
```

The first item in the log entry is the date and time of the message. The next is the module producing the message (core, in this case) and the severity level of that message. This is followed by the process ID and, if appropriate, the thread ID, of the process that experienced the condition. Next, we have the client address that made the request. And finally is the detailed error message, which in this case indicates a request for a file that did not exist.

A very wide variety of different messages can appear in the error log. Most look similar to the example above. The error log will also contain debugging output from CGI scripts. Any information written to `stderr` by a CGI script will be copied directly to the error log.

Putting a `%L` token in both the error log and the access log will produce a log entry ID with which you can correlate the entry in the error log with the entry in the access log. If mod_unique_id is loaded, its unique request ID will be used as the log entry ID, too.

During testing, it is often useful to continuously monitor the error log for any problems. On Unix systems, you can accomplish this using:

```
tail -f error_log
```

Per-module logging

The LogLevel directive allows you to specify a log severity level on a per-module basis. In this way, if you are troubleshooting a problem with just one particular module, you can turn up its logging volume without also getting the details of other modules that you're not interested in. This is particularly useful for modules such as mod_proxy or mod_rewrite where you want to know details about what it's trying to do.

Do this by specifying the name of the module in your `LogLevel` directive:

```
LogLevel info rewrite:trace5
```

This sets the main `LogLevel` to info, but turns it up to `trace5` for mod_rewrite.

This replaces the per-module logging directives, such as `RewriteLog`, that were present in earlier versions of the server.

Access Log

Related Modules	Related Directives
mod_log_config	CustomLog
mod_setenvif	LogFormat
	SetEnvIf

The server access log records all requests processed by the server. The location and content of the access log are controlled by the CustomLog directive. The LogFormat directive can be used to simplify the selection of the contents of the logs. This section describes how to configure the server to record information in the access log.

Of course, storing the information in the access log is only the start of log management. The next step is to analyze this information to produce useful statistics. Log analysis in general is beyond the scope of this document, and not really part of the job of the web server itself. For more information about this topic, and for applications which perform log analysis, check the Open Directory.

Various versions of Apache httpd have used other modules and directives to control access logging, including mod_log_referer, mod_log_agent, and the `TransferLog` directive. The `CustomLog` directive now subsumes the functionality of all the older directives.

The format of the access log is highly configurable. The format is specified using a format string that looks much like a C-style printf(1) format string. Some examples are presented in the next sections. For a complete list of the possible contents of the format string, see the `mod_log_config` format strings.

Common Log Format

A typical configuration for the access log might look as follows.

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog "logs/access_log" common
```

This defines the *nickname* `common` and associates it with a particular log format string. The format string consists of percent directives, each of which tell the server to log a particular piece of information. Literal characters may also be placed in the format string and will be copied directly into the log output. The quote character (`"`) must be escaped by placing a backslash before it to prevent it from being interpreted as the end of the format string. The format string may also contain the special control characters "`\n`" for new-line and "`\t`" for tab.

The `CustomLog` directive sets up a new log file using the defined *nickname*. The filename for the access log is relative to the `ServerRoot` unless it begins with a slash.

The above configuration will write log entries in a format known as the Common Log Format (CLF). This standard format can be produced by many different web servers and read by many log analysis programs. The log file entries produced in CLF will look something like this:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET
/apache_pb.gif HTTP/1.0" 200 2326
```

Each part of this log entry is described below.

`127.0.0.1` (`%h`)
: This is the IP address of the client (remote host) which made the request to the server. If `HostnameLookups` is set to `On`, then the server will try to determine the hostname and log it in place of the IP address. However, this configuration is not recommended since it can significantly slow the server. Instead, it is best to use a log post-processor such as `logresolve` to determine the hostnames. The IP address reported here is not necessarily the address of the machine at which the user is sitting. If a proxy server exists between the user and the server, this address will be the address of the proxy, rather than the originating machine.

`-` (`%l`)
: The "hyphen" in the output indicates that the requested piece of information is not available. In this case, the information that is not available is the RFC 1413 identity of the client determined by `identd` on the clients machine.

This information is highly unreliable and should almost never be used except on tightly controlled internal networks. Apache httpd will not even attempt to determine this information unless `IdentityCheck` is set to `On`.

`frank` (`%u`)

This is the userid of the person requesting the document as determined by HTTP authentication. The same value is typically provided to CGI scripts in the `REMOTE_USER` environment variable. If the status code for the request (see below) is 401, then this value should not be trusted because the user is not yet authenticated. If the document is not password protected, this part will be "`-`" just like the previous one.

`[10/Oct/2000:13:55:36 -0700]` (`%t`)

The time that the request was received. The format is:

```
[day/month/year:hour:minute:second zone]
day = 2*digit
month = 3*letter
year = 4*digit
hour = 2*digit
minute = 2*digit
second = 2*digit
zone = (`+' | `-') 4*digit
```

It is possible to have the time displayed in another format by specifying `%{format}t` in the log format string, where `format` is either as in `strftime(3)` from the C standard library, or one of the supported special tokens. For details see the `mod_log_config` format strings.

`"GET /apache_pb.gif HTTP/1.0"` (`\"%r\"`)

The request line from the client is given in double quotes. The request line contains a great deal of useful information. First, the method used by the client is `GET`. Second, the client requested the resource `/apache_pb.gif`, and third, the client used the protocol `HTTP/1.0`. It is also possible to log one or more parts of the request line independently. For example, the format string "`%m %U%q %H`" will log the method, path, query-string, and protocol, resulting in exactly the same output as "`%r`".

`200` (`%>s`)

This is the status code that the server sends back to the client. This information is very valuable, because it reveals whether the request resulted in a successful response (codes beginning in 2), a redirection (codes beginning in 3), an error caused by the client (codes beginning in 4), or an error in the server (codes beginning in 5). The full list of possible status codes can be found in the HTTP specification (RFC2616 section 10).

`2326` (`%b`)

The last part indicates the size of the object returned to the client, not including the response headers. If no content was returned to the client, this value will be "`-`". To log "`0`" for no content, use `%B` instead.

Combined Log Format

Another commonly used format string is called the Combined Log Format. It can be used as follows.

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \'
CustomLog "log/access_log" combined
```

This format is exactly the same as the Common Log Format, with the addition of two more fields. Each of the additional fields uses the percent-directive `%{`*`header`*`}i`, where *header* can be any HTTP request header. The access log under this format will look like:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET
/apache_pb.gif HTTP/1.0" 200 2326
"http://www.example.com/start.html" "Mozilla/4.08 [en]
(Win98; I ;Nav)"
```

The additional fields are:

`"http://www.example.com/start.html"` (`\"%{Referer}i\"`)
: The "Referer" (sic) HTTP request header. This gives the site that the client reports having been referred from. (This should be the page that links to or includes `/apache_pb.gif`).

`"Mozilla/4.08 [en] (Win98; I ;Nav)"` (`\"%{User-agent}i\"`)
: The User-Agent HTTP request header. This is the identifying information that the client browser reports about itself.

Multiple Access Logs

Multiple access logs can be created simply by specifying multiple `CustomLog` directives in the configuration file. For example, the following directives will create three access logs. The first contains the basic CLF information, while the second and third contain referer and browser information. The last two `CustomLog` lines show how to mimic the effects of the `ReferLog` and `AgentLog` directives.

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog "logs/access_log" common
CustomLog "logs/referer_log" "%{Referer}i -> %U"
CustomLog "logs/agent_log" "%{User-agent}i"
```

This example also shows that it is not necessary to define a nickname with the `LogFormat` directive. Instead, the log format can be specified directly in the `CustomLog` directive.

Conditional Logs

There are times when it is convenient to exclude certain entries from the access logs based on characteristics of the client request. This is easily accomplished with the help of environment variables. First, an environment variable must be set to indicate that the request meets certain conditions. This is usually accomplished with `SetEnvIf`. Then the `env=` clause of the `CustomLog` directive is used to include or exclude requests where the environment variable is set. Some examples:

```
# Mark requests from the loop-back interface
SetEnvIf Remote_Addr "127\.0\.0\.1" dontlog
# Mark requests for the robots.txt file
SetEnvIf Request_URI "^/robots\.txt$" dontlog
# Log what remains
CustomLog "logs/access_log" common env=!dontlog
```

As another example, consider logging requests from english-speakers to one log file, and non-english speakers to a different log file.

```
SetEnvIf Accept-Language "en" english
CustomLog "logs/english_log" common env=english
CustomLog "logs/non_english_log" common env=!english
```

In a caching scenario one would want to know about the efficiency of the cache. A very simple method to find this out would be:

```
SetEnv CACHE_MISS 1
LogFormat "%h %l %u %t "%r " %>s %b %{CACHE_MISS}e" con
CustomLog "logs/access_log" common-cache
```

mod_cache will run before mod_env and, when successful, will deliver the content without it. In that case a cache hit will log -, while a cache miss will log 1.

In addition to the env= syntax, LogFormat supports logging values conditional upon the HTTP response code:

```
LogFormat "%400,501{User-agent}i" browserlog
LogFormat "%!200,304,302{Referer}i" refererlog
```

In the first example, the User-agent will be logged if the HTTP status code is 400 or 501. In other cases, a literal "-" will be logged instead. Likewise, in the second example, the Referer will be logged if the HTTP status code is **not** 200, 204, or 302. (Note the "!" before the status codes.

Although we have just shown that conditional logging is very powerful and flexible, it is not the only way to control the contents of the logs. Log files are more useful when they contain a complete record of server activity. It is often easier to simply post-process the log files to remove requests that you do not want to consider.

Log Rotation

On even a moderately busy server, the quantity of information stored in the log files is very large. The access log file typically grows 1 MB or more per 10,000 requests. It will consequently be necessary to periodically rotate the log files by moving or deleting the existing logs. This cannot be done while the server is running, because Apache httpd will continue writing to the old log file as long as it holds the file open. Instead, the server must be restarted after the log files are moved or deleted so that it will open new log files.

By using a *graceful* restart, the server can be instructed to open new log files without losing any existing or pending connections from clients. However, in order to accomplish this, the server must continue to write to the old log files while it finishes serving old requests. It is therefore necessary to wait for some time after the restart before doing any processing on the log files. A typical scenario that simply rotates the logs and compresses the old logs to save space is:

```
mv access_log access_log.old
mv error_log error_log.old
apachectl graceful
sleep 600
gzip access_log.old error_log.old
```

Another way to perform log rotation is using piped logs as discussed in the next section.

Piped Logs

Apache httpd is capable of writing error and access log files through a pipe to another process, rather than directly to a file. This capability dramatically increases the flexibility of logging, without adding code to the main server. In order to write logs to a pipe, simply replace the filename with the pipe character "|", followed by the name of the executable which should accept log entries on its standard input. The server will start the piped-log process when the server starts, and will restart it if it crashes while the server is running. (This last feature is why we can refer to this technique as "reliable piped logging".)

Piped log processes are spawned by the parent Apache httpd process, and inherit the userid of that process. This means that piped log programs usually run as root. It is therefore very important to keep the programs simple and secure.

One important use of piped logs is to allow log rotation without having to restart the server. The Apache HTTP Server includes a simple program called rotatelogs for this purpose. For example, to rotate the logs every 24 hours, you can use:

```
CustomLog "|/usr/local/apache/bin/rotatelogs /var/log/a
```

Notice that quotes are used to enclose the entire command that will be called for the pipe. Although these examples are for the access log, the same technique can be used for the error log.

As with conditional logging, piped logs are a very powerful tool, but they should not be used where a simpler solution like off-line post-processing is available.

By default the piped log process is spawned without invoking a shell. Use "`|$`" instead of "`|`" to spawn using a shell (usually with `/bin/sh -c`):

```
# Invoke "rotatelogs" using a shell
CustomLog "|$/usr/local/apache/bin/rotatelogs   /var/lo
```

This was the default behaviour for Apache 2.2. Depending on the shell specifics this might lead to an additional shell process for the lifetime of the logging pipe program and signal handling problems during restart. For compatibility reasons with Apache 2.2 the notation "`||`" is also supported and equivalent to using "`|`".

Windows note

Note that on Windows, you may run into problems when running many piped logger processes, especially when HTTPD is running as a service. This is caused by running out of desktop heap space. The desktop heap space given to each service is specified by the third argument to the `SharedSection` parameter in the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager\SubSystems\Windows registry value. **Change this value with care**; the normal caveats for changing the Windows registry apply, but you might also exhaust the desktop heap pool if the number is adjusted too high.

Virtual Hosts

When running a server with many virtual hosts, there are several options for dealing with log files. First, it is possible to use logs exactly as in a single-host server. Simply by placing the logging directives outside the `<VirtualHost>` sections in the main server context, it is possible to log all requests in the same access log and error log. This technique does not allow for easy collection of statistics on individual virtual hosts.

If `CustomLog` or `ErrorLog` directives are placed inside a `<VirtualHost>` section, all requests or errors for that virtual host will be logged only to the specified file. Any virtual host which does not have logging directives will still have its requests sent to the main server logs. This technique is very useful for a small number of virtual hosts, but if the number of hosts is very large, it can be complicated to manage. In addition, it can often create problems with insufficient file descriptors.

For the access log, there is a very good compromise. By adding information on the virtual host to the log format string, it is possible to log all hosts to the same log, and later split the log into individual files. For example, consider the following directives.

```
LogFormat "%v %l %u %t \"%r\" %>s %b" comonvhost
CustomLog "logs/access_log" comonvhost
```

The `%v` is used to log the name of the virtual host that is serving the request. Then a program like split-logfile can be used to post-process the access log in order to split it into one file per virtual host.

Other Log Files

Related Modules	Related Directives
mod_logio	LogFormat
mod_log_config	BufferedLogs
mod_log_forensic	ForensicLog
mod_cgi	PidFile
	ScriptLog
	ScriptLogBuffer
	ScriptLogLength

Logging actual bytes sent and received

mod_logio adds in two additional LogFormat fields (%I and %O) that log the actual number of bytes received and sent on the network.

Forensic Logging

mod_log_forensic provides for forensic logging of client requests. Logging is done before and after processing a request, so the forensic log contains two log lines for each request. The forensic logger is very strict with no customizations. It can be an invaluable debugging and security tool.

PID File

On startup, Apache httpd saves the process id of the parent httpd process to the file `logs/httpd.pid`. This filename can be changed with the PidFile directive. The process-id is for use by the administrator in restarting and terminating the daemon by sending signals to the parent process; on Windows, use the -k command line option instead. For more information see the Stopping and Restarting page.

Script Log

In order to aid in debugging, the ScriptLog directive allows you to record the input to and output from CGI scripts. This should only be used in testing - not for live servers. More information is available in the mod_cgi documentation.