Apache Module mod_rewrite

Description:	Provides a rule-based rewriting engine to rewrite requested URLs on the fly
Status:	Extension
Module Identifier:	rewrite_module
Source File:	mod_rewrite.c
Compatibility:	Available in Apache 1.3 and later

Summary

``The great thing about mod_rewrite is it gives you all the configurability and flexibility of Sendmail. The downside to mod_rewrite is that it gives you all the configurability and flexibility of Sendmail.''

-- Brian Behlendorf
Apache Group

`` Despite the tons of examples and docs, mod_rewrite is voodoo. Damned cool voodoo, but still voodoo. ''

-- Brian Moore
bem@news.cmc.net

Welcome to mod_rewrite, the Swiss Army Knife of URL manipulation!

This module uses a rule-based rewriting engine (based on a regular-expression parser) to rewrite requested URLs on the fly. It supports an unlimited number of rules and an unlimited number of attached rule conditions for each rule to provide a really flexible and powerful URL manipulation mechanism. The URL manipulations can depend on various tests, for instance server variables, environment variables, HTTP headers, time stamps and even external database lookups in various formats can be used to achieve a really granular URL matching.

This module operates on the full URLs (including the path-info part) both in per-server context (`httpd.conf`) and per-directory context (`.htaccess`) and can even generate query-string parts on result. The rewritten result can lead to internal sub-processing, external request redirection or even to an internal proxy throughput.

But all this functionality and flexibility has its drawback: complexity. So don't expect to understand this entire module in just one day.

This module was invented and originally written in April 1996 and gifted exclusively to the The Apache Group in July 1997 by

`Ralf S. Engelschall`[1]
`rse@engelschall.com`
`www.engelschall.com`[1]

Topics

Directives

Internal Processing

The internal processing of this module is very complex but needs to be explained once even to the average user to avoid common mistakes and to let you exploit its full functionality.

API Phases

First you have to understand that when Apache processes a HTTP request it does this in phases. A hook for each of these phases is provided by the Apache API. Mod_rewrite uses two of these hooks: the URL-to-filename translation hook which is used after the HTTP request has been read but before any authorization starts and the Fixup hook which is triggered after the authorization phases and after the per-directory config files (`.htaccess`) have been read, but before the content handler is activated.

So, after a request comes in and Apache has determined the corresponding server (or virtual server) the rewriting engine starts processing of all mod_rewrite directives from the per-server configuration in the URL-to-filename phase. A few steps later when the final data directories are found, the per-directory configuration directives of mod_rewrite are triggered in the Fixup phase. In both situations mod_rewrite rewrites URLs either to new URLs or to filenames, although there is no obvious distinction between them. This is a usage of the API which was not intended to be this way when the API was designed, but as of Apache 1.x this is the only way mod_rewrite can operate. To make this point more clear remember the following two points:

1. Although mod_rewrite rewrites URLs to URLs, URLs to filenames and even filenames to filenames, the API currently provides only a URL-to-filename hook. In Apache 2.0 the two missing hooks will be added to make the processing more clear. But this point has no drawbacks for the user, it is just a fact which should be remembered: Apache does more in the URL-to-filename hook than the API intends for it.
2. Unbelievably mod_rewrite provides URL manipulations in per-directory context, *i.e.*, within `.htaccess` files, although these are reached a very long time after the URLs have been translated to filenames. It has to be this way because `.htaccess` files live in the filesystem, so processing has already reached this stage. In other words: According to the API phases at this time it is too late for any URL manipulations. To overcome this chicken and egg problem mod_rewrite uses a trick: When you manipulate a URL/filename in per-directory context mod_rewrite first rewrites the filename back to its corresponding URL (which is usually impossible, but see the `RewriteBase` directive below for the trick to achieve this) and then initiates a new internal sub-request with the new URL. This restarts processing of the API phases.

   Again mod_rewrite tries hard to make this complicated step totally transparent to the user, but you should remember here: While URL manipulations in per-server context are really fast and efficient, per-directory rewrites are slow and inefficient due to this chicken and egg problem. But on the other hand this is the only way mod_rewrite can provide (locally restricted) URL manipulations to the average user.

Don't forget these two points!

Ruleset Processing

Now when mod_rewrite is triggered in these two API phases, it reads the configured rulesets from its configuration structure (which itself was either created on startup for per-server context or during the directory walk of the Apache kernel for per-directory context). Then the URL rewriting engine is started with the contained ruleset (one or more rules together with their conditions). The operation of the URL rewriting engine itself is exactly the same for both configuration contexts. Only the final result processing is different.

The order of rules in the ruleset is important because the rewriting engine processes them in a special (and not very obvious) order. The rule is this: The rewriting engine loops through the ruleset rule by rule (`RewriteRule` directives) and when a particular rule matches it optionally loops through existing corresponding conditions (`RewriteCond` directives). For historical reasons the conditions are given first, and so the control flow is a little bit long-winded. See Figure 1 for more details.

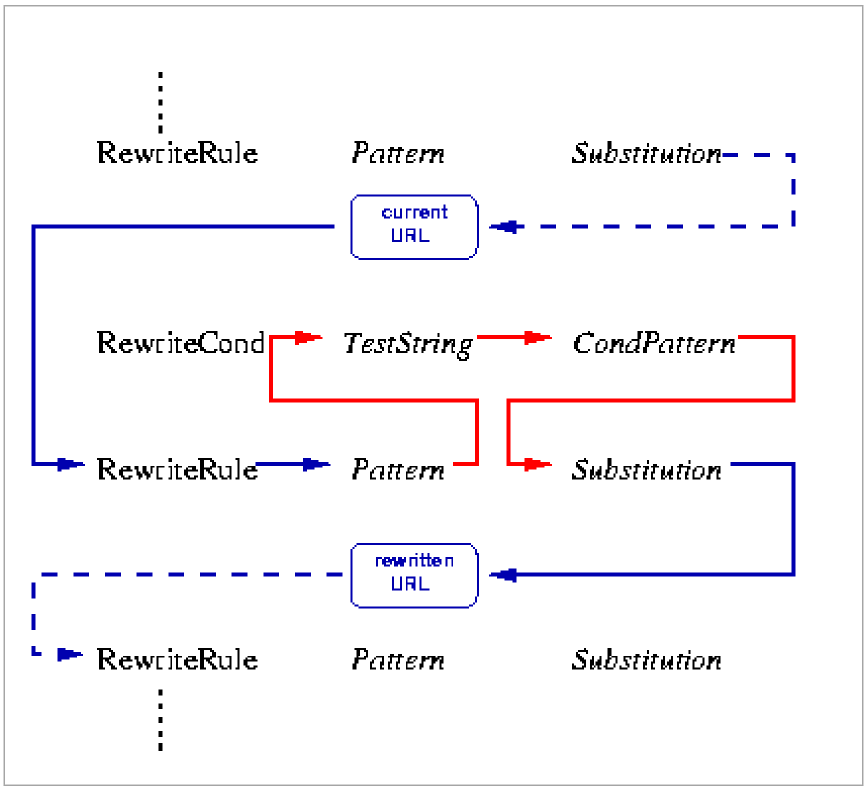

Figure 1:*The control flow through the rewriting ruleset*

As you can see, first the URL is matched against the *Pattern* of each rule. When it fails mod_rewrite immediately stops processing this rule and continues with the next rule. If the *Pattern* matches, mod_rewrite looks for corresponding rule conditions. If none are present, it just substitutes the URL with a new value which is constructed from the string *Substitution* and goes on with its rule-looping. But if conditions exist, it starts an inner loop for processing them in the order that they are listed. For conditions the logic is different: we don't match a pattern against the current URL. Instead we first create a string *TestString* by expanding variables, back-references, map lookups, *etc.* and then we try to match *CondPattern* against it. If the pattern doesn't match, the complete set of conditions and the corresponding rule fails. If the pattern matches, then the next condition is processed until no more conditions are available. If all conditions match, processing is continued with the substitution of the URL with *Substitution*.

Quoting Special Characters

As of Apache 1.3.20, special characters in *TestString* and *Substitution* strings can be escaped (that is, treated as normal characters without their usual special meaning) by prefixing them with a slosh ('\') character. In other words, you can include an actual dollar-sign character in a *Substitution* string by using '\$'; this keeps mod_rewrite from trying to treat it as a backreference.

Regex Back-Reference Availability

One important thing here has to be remembered: Whenever you use parentheses in *Pattern* or in one of the *CondPattern*, back-references are internally created which can be used with the strings `$N` and `%N` (see below). These are available for creating the strings *Substitution* and *TestString*. Figure 2 shows to which locations the back-references are transfered for expansion.

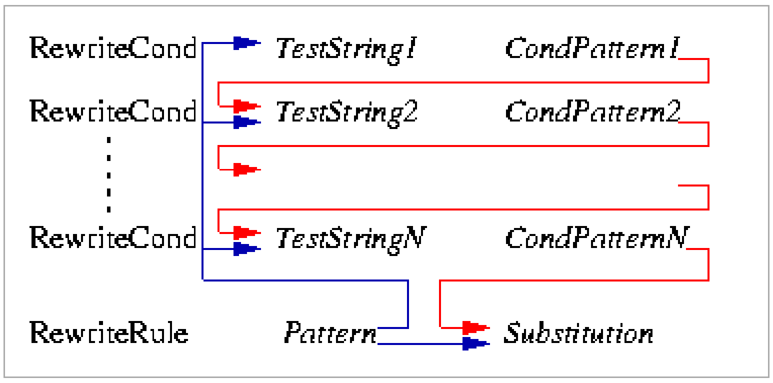

Figure 2: *The back-reference flow through a rule.*

We know this was a crash course on mod_rewrite's internal processing. But you will benefit from this knowledge when reading the following documentation of the available directives.

Environment Variables

This module keeps track of two additional (non-standard) CGI/SSI environment variables named `SCRIPT_URL` and `SCRIPT_URI`. These contain the *logical* Web-view to the current resource, while the standard CGI/SSI variables `SCRIPT_NAME` and `SCRIPT_FILENAME` contain the *physical* System-view.

Notice: These variables hold the URI/URL *as they were initially requested*, *i.e.*, *before* any rewriting. This is important because the rewriting process is primarily used to rewrite logical URLs to physical pathnames.

Example

```
SCRIPT_NAME=/sw/lib/w3s/tree/global/u/rse/.www/index.html
SCRIPT_FILENAME=/u/rse/.www/index.html
SCRIPT_URL=/u/rse/
SCRIPT_URI=http://en1.engelschall.com/u/rse/
```

Practical Solutions

We also have an URL Rewriting Guide[3] available, which provides a collection of practical solutions for URL-based problems. There you can find real-life rulesets and additional information about mod_rewrite.

RewriteBase Directive

Description:	Sets the base URL for per-directory rewrites
Syntax:	`RewriteBase` *`URL-path`*
Default:	`See usage for information.`
Context:	directory, .htaccess
Override:	FileInfo
Status:	Extension
Module:	mod_rewrite

The `RewriteBase` directive explicitly sets the base URL for per-directory rewrites. As you will see below, `RewriteRule` can be used in per-directory config files (`.htaccess`). There it will act locally, *i.e.*, the local directory prefix is stripped at this stage of processing and your rewriting rules act only on the remainder. At the end it is automatically added back to the path. The default setting is; `RewriteBase` *physical-directory-path*

When a substitution occurs for a new URL, this module has to re-inject the URL into the server processing. To be able to do this it needs to know what the corresponding URL-prefix or URL-base is. By default this prefix is the corresponding filepath itself. **But at most websites URLs are NOT directly related to physical filename paths, so this assumption will usually be wrong!** There you have to use the `RewriteBase` directive to specify the correct URL-prefix.

If your webserver's URLs are **not** directly related to physical file paths, you have to use `RewriteBase` in every `.htaccess` files where you want to use `RewriteRule` directives.

For example, assume the following per-directory config file:

```
#
#  /abc/def/.htaccess -- per-dir config file for directory /abc/def
#  Remember: /abc/def is the physical path of /xyz, i.e., the server
#            has a 'Alias /xyz /abc/def' directive e.g.
#

RewriteEngine On

#  let the server know that we were reached via /xyz and not
#  via the physical path prefix /abc/def
RewriteBase   /xyz

#  now the rewriting rules
RewriteRule   ^oldstuff\.html$  newstuff.html
```

In the above example, a request to `/xyz/oldstuff.html` gets correctly rewritten to the physical file `/abc/def/newstuff.html`.

For Apache Hackers

The following list gives detailed information about the internal processing steps:

```
Request:
  /xyz/oldstuff.html
```

```
Internal Processing:
  /xyz/oldstuff.html     -> /abc/def/oldstuff.html  (per-server Alias)
  /abc/def/oldstuff.html -> /abc/def/newstuff.html  (per-dir    RewriteRule)
  /abc/def/newstuff.html -> /xyz/newstuff.html      (per-dir    RewriteBase)
  /xyz/newstuff.html     -> /abc/def/newstuff.html  (per-server Alias)

Result:
  /abc/def/newstuff.html
```

This seems very complicated but is the correct Apache internal processing, because the per-directory rewriting comes too late in the process. So, when it occurs the (rewritten) request has to be re-injected into the Apache kernel! BUT: While this seems like a serious overhead, it really isn't, because this re-injection happens fully internally to the Apache server and the same procedure is used by many other operations inside Apache. So, you can be sure the design and implementation is correct.

RewriteCond Directive

Description:	Defines a condition under which rewriting will take place
Syntax:	`RewriteCond` *`TestString CondPattern`*
Default:	`None`
Context:	server config, virtual host, directory, .htaccess
Override:	FileInfo
Status:	Extension
Module:	mod_rewrite

The `RewriteCond` directive defines a rule condition. Precede a `RewriteRule` directive with one or more `RewriteCond` directives. The following rewriting rule is only used if its pattern matches the current state of the URI **and** if these additional conditions apply too.

TestString is a string which can contains the following expanded constructs in addition to plain text:

- **RewriteRule backreferences**: These are backreferences of the form

  `$N`

  (0 <= N <= 9) which provide access to the grouped parts (parenthesis!) of the pattern from the corresponding `RewriteRule` directive (the one following the current bunch of `RewriteCond` directives).
- **RewriteCond backreferences**: These are backreferences of the form

  `%N`

  (1 <= N <= 9) which provide access to the grouped parts (parentheses!) of the pattern from the last matched `RewriteCond` directive in the current bunch of conditions.
- **RewriteMap expansions**: These are expansions of the form

  `${mapname:key|default}`

  See the documentation for RewriteMap for more details.
- **Server-Variables**: These are variables of the form

  `%{` ***NAME_OF_VARIABLE*** **`}`**

where *NAME_OF_VARIABLE* can be a string taken from the following list:

HTTP headers:	**connection & request:**	
HTTP_USER_AGENT	REMOTE_ADDR	
HTTP_REFERER	REMOTE_HOST	
HTTP_COOKIE	REMOTE_USER	
HTTP_FORWARDED	REMOTE_IDENT	
HTTP_HOST	REQUEST_METHOD	
HTTP_PROXY_CONNECTION	SCRIPT_FILENAME	
HTTP_ACCEPT	PATH_INFO	
	QUERY_STRING	
	AUTH_TYPE	
server internals:	**system stuff:**	**specials:**
DOCUMENT_ROOT	TIME_YEAR	API_VERSION
SERVER_ADMIN	TIME_MON	THE_REQUEST
SERVER_NAME	TIME_DAY	REQUEST_URI
SERVER_ADDR	TIME_HOUR	REQUEST_FILENAME
SERVER_PORT	TIME_MIN	IS_SUBREQ
SERVER_PROTOCOL	TIME_SEC	
SERVER_SOFTWARE	TIME_WDAY	
	TIME	

These variables all correspond to the similarly named HTTP MIME-headers, C variables of the Apache server or `struct tm` fields of the Unix system. Most are documented elsewhere in the Manual or in the CGI specification. Those that are special to mod_rewrite include:

`IS_SUBREQ`
: Will contain the text "true" if the request currently being processed is a sub-request, "false" otherwise. Sub-requests may be generated by modules that need to resolve additional files or URIs in order to complete their tasks.

`API_VERSION`
: This is the version of the Apache module API (the internal interface between server and module) in the current httpd build, as defined in include/ap_mmn.h. The module API version corresponds to the version of Apache in use (in the release version of Apache 1.3.14, for instance, it is 19990320:10), but is mainly of interest to module authors.

`THE_REQUEST`
: The full HTTP request line sent by the browser to the server (e.g., "`GET /index.html HTTP/1.1`"). This does not include any additional headers sent by the browser.

`REQUEST_URI`
: The resource requested in the HTTP request line. (In the example above, this would be "/index.html".)

`REQUEST_FILENAME`
: The full local filesystem path to the file or script matching the request.

Special Notes:

1. The variables SCRIPT_FILENAME and REQUEST_FILENAME contain the same value, *i.e.*, the value of the `filename` field of the internal `request_rec` structure of the Apache server.

The first name is just the commonly known CGI variable name while the second is the consistent counterpart to REQUEST_URI (which contains the value of the `uri` field of `request_rec`).

2. There is the special format: `%{ENV:variable}` where *variable* can be any environment variable. This is looked-up via internal Apache structures and (if not found there) via `getenv()` from the Apache server process.
3. There is the special format: `%{HTTP:header}` where *header* can be any HTTP MIME-header name. This is looked-up from the HTTP request. Example: `%{HTTP:Proxy-Connection}` is the value of the HTTP header ``` ``Proxy-Connection:'' ```.
4. There is the special format `%{LA-U:variable}` for look-aheads which perform an internal (URL-based) sub-request to determine the final value of *variable*. Use this when you want to use a variable for rewriting which is actually set later in an API phase and thus is not available at the current stage. For instance when you want to rewrite according to the `REMOTE_USER` variable from within the per-server context (`httpd.conf` file) you have to use `%{LA-U:REMOTE_USER}` because this variable is set by the authorization phases which come *after* the URL translation phase where mod_rewrite operates. On the other hand, because mod_rewrite implements its per-directory context (`.htaccess` file) via the Fixup phase of the API and because the authorization phases come *before* this phase, you just can use `%{REMOTE_USER}` there.
5. There is the special format: `%{LA-F:variable}` which performs an internal (filename-based) sub-request to determine the final value of *variable*. Most of the time this is the same as LA-U above.

CondPattern is the condition pattern, *i.e.*, a regular expression which is applied to the current instance of the *TestString*, *i.e.*, *TestString* is evaluated and then matched against *CondPattern*.

Remember: *CondPattern* is a *perl compatible regular expression* with some additions:

1. You can prefix the pattern string with a `'!'` character (exclamation mark) to specify a **non**-matching pattern.
2. There are some special variants of *CondPatterns*. Instead of real regular expression strings you can also use one of the following:
   - '**<CondPattern**' (is lexically lower)
     Treats the *CondPattern* as a plain string and compares it lexically to *TestString*. True if *TestString* is lexically lower than *CondPattern*.
   - '**>CondPattern**' (is lexically greater)
     Treats the *CondPattern* as a plain string and compares it lexically to *TestString*. True if *TestString* is lexically greater than *CondPattern*.
   - '**=CondPattern**' (is lexically equal)
     Treats the *CondPattern* as a plain string and compares it lexically to *TestString*. True if *TestString* is lexically equal to *CondPattern*, i.e the two strings are exactly equal (character by character). If *CondPattern* is just `""` (two quotation marks) this compares *TestString* to the empty string.
   - '**-d**' (is **d**irectory)
     Treats the *TestString* as a pathname and tests if it exists and is a directory.
   - '**-f**' (is regular **f**ile)
     Treats the *TestString* as a pathname and tests if it exists and is a regular file.
   - '**-s**' (is regular file with **s**ize)
     Treats the *TestString* as a pathname and tests if it exists and is a regular file with size greater than zero.
   - '**-l**' (is symbolic **l**ink)
     Treats the *TestString* as a pathname and tests if it exists and is a symbolic link.

- '**-F**' (is existing file via subrequest)
  Checks if *TestString* is a valid file and accessible via all the server's currently-configured access controls for that path. This uses an internal subrequest to determine the check, so use it with care because it decreases your servers performance!
- '**-U**' (is existing URL via subrequest)
  Checks if *TestString* is a valid URL and accessible via all the server's currently-configured access controls for that path. This uses an internal subrequest to determine the check, so use it with care because it decreases your server's performance!

Notice

All of these tests can also be prefixed by an exclamation mark ('!') to negate their meaning.

Additionally you can set special flags for *CondPattern* by appending

[*flags*]

as the third argument to the `RewriteCond` directive. *Flags* is a comma-separated list of the following flags:

- '**`nocase|NC`**' (**n**o **c**ase)
  This makes the test case-insensitive, *i.e.*, there is no difference between 'A-Z' and 'a-z' both in the expanded *TestString* and the *CondPattern*. This flag is effective only for comparisons between *TestString* and *CondPattern*. It has no effect on filesystem and subrequest checks.
- '**`ornext|OR`**' (**or** next condition)
  Use this to combine rule conditions with a local OR instead of the implicit AND. Typical example:

```
RewriteCond %{REMOTE_HOST}  ^host1.*  [OR]
RewriteCond %{REMOTE_HOST}  ^host2.*  [OR]
RewriteCond %{REMOTE_HOST}  ^host3.*
RewriteRule ...some special stuff for any of these hosts...
```

  Without this flag you would have to write the cond/rule three times.

Example:

To rewrite the Homepage of a site according to the ``User-Agent:'' header of the request, you can use the following:

```
RewriteCond  %{HTTP_USER_AGENT}  ^Mozilla.*
RewriteRule  ^/$                 /homepage.max.html  [L]

RewriteCond  %{HTTP_USER_AGENT}  ^Lynx.*
RewriteRule  ^/$                 /homepage.min.html  [L]

RewriteRule  ^/$                 /homepage.std.html  [L]
```

Interpretation: If you use Netscape Navigator as your browser (which identifies itself as 'Mozilla'), then you get the max homepage, which includes Frames, *etc.* If you use the Lynx browser (which is

Terminal-based), then you get the min homepage, which contains no images, no tables, *etc*. If you use any other browser you get the standard homepage.

RewriteEngine Directive

Description:	Enables or disables runtime rewriting engine
Syntax:	`RewriteEngine on|off`
Default:	`RewriteEngine off`
Context:	server config, virtual host, directory, .htaccess
Override:	FileInfo
Status:	Extension
Module:	mod_rewrite

The `RewriteEngine` directive enables or disables the runtime rewriting engine. If it is set to `off` this module does no runtime processing at all. It does not even update the `SCRIPT_URx` environment variables.

Use this directive to disable the module instead of commenting out all the `RewriteRule` directives!

Note that, by default, rewrite configurations are not inherited. This means that you need to have a `RewriteEngine on` directive for each virtual host in which you wish to use it.

RewriteLock Directive

Description:	Sets the name of the lock file used for RewriteMap synchronization
Syntax:	`RewriteLock` *`file-path`*
Default:	`None`
Context:	server config
Status:	Extension
Module:	mod_rewrite

This directive sets the filename for a synchronization lockfile which mod_rewrite needs to communicate with `RewriteMap` *programs*. Set this lockfile to a local path (not on a NFS-mounted device) when you want to use a rewriting map-program. It is not required for other types of rewriting maps.

RewriteLog Directive

Description:	Sets the name of the file used for logging rewrite engine processing
Syntax:	`RewriteLog` *`file-path`*
Context:	server config, virtual host
Status:	Extension
Module:	mod_rewrite

The `RewriteLog` directive sets the name of the file to which the server logs any rewriting actions it performs. If the name does not begin with a slash ('`/`') then it is assumed to be relative to the *Server Root*. The directive should occur only once per server config.

To disable the logging of rewriting actions it is not recommended to set *Filename* to `/dev/null`, because although the rewriting engine does not then output to a logfile it still creates the logfile output internally. **This will slow down the server with no advantage to the administrator!** To disable logging either remove or comment out the `RewriteLog` directive or use `RewriteLogLevel 0`!

Security

See the Apache Security Tips[4] document for details on why your security could be compromised if the directory where logfiles are stored is writable by anyone other than the user that starts the server.

Example

```
RewriteLog "/usr/local/var/apache/logs/rewrite.log"
```

RewriteLogLevel Directive

Description:	Sets the verbosity of the log file used by the rewrite engine
Syntax:	`RewriteLogLevel` *`Level`*
Default:	`RewriteLogLevel 0`
Context:	server config, virtual host
Status:	Extension
Module:	mod_rewrite

The `RewriteLogLevel` directive sets the verbosity level of the rewriting logfile. The default level 0 means no logging, while 9 or more means that practically all actions are logged.

To disable the logging of rewriting actions simply set *Level* to 0. This disables all rewrite action logs.

Using a high value for *Level* will slow down your Apache server dramatically! Use the rewriting logfile at a *Level* greater than 2 only for debugging!

Example

```
RewriteLogLevel 3
```

RewriteMap Directive

Description:	Defines a mapping function for key-lookup
Syntax:	`RewriteMap` *`MapName MapType`*`:`*`MapSource`*
Default:	`None`
Context:	server config, virtual host
Status:	Extension
Module:	mod_rewrite
Compatibility:	The choice of different dbm types is available in Apache 2.0.41 and later

The `RewriteMap` directive defines a *Rewriting Map* which can be used inside rule substitution strings by the mapping-functions to insert/substitute fields through a key lookup. The source of this lookup can be of various types.

The *MapName* is the name of the map and will be used to specify a mapping-function for the substitution strings of a rewriting rule via one of the following constructs:

${ *MapName* : *LookupKey* }
${ *MapName* : *LookupKey* | *DefaultValue* }

When such a construct occurs the map *MapName* is consulted and the key *LookupKey* is looked-up. If the key is found, the map-function construct is substituted by *SubstValue*. If the key is not found then it is substituted by *DefaultValue* or by the empty string if no *DefaultValue* was specified.

The following combinations for *MapType* and *MapSource* can be used:

- **Standard Plain Text**
  MapType: `txt`, MapSource: Unix filesystem path to valid regular file

  This is the standard rewriting map feature where the *MapSource* is a plain ASCII file containing either blank lines, comment lines (starting with a '#' character) or pairs like the following - one per line.

  MatchingKey SubstValue

  Example

  ```
  ##
  ##  map.txt -- rewriting map
  ##

  Ralf.S.Engelschall    rse   # Bastard Operator From Hell
  Mr.Joe.Average        joe   # Mr. Average
  ```

  ```
  RewriteMap real-to-user txt:/path/to/file/map.txt
  ```

- **Randomized Plain Text**
  MapType: `rnd`, MapSource: Unix filesystem path to valid regular file

  This is identical to the Standard Plain Text variant above but with a special post-processing feature: After looking up a value it is parsed according to contained ``|'' characters which have the meaning of ``or''. In other words they indicate a set of alternatives from which the actual returned value is chosen randomly. Although this sounds crazy and useless, it was actually designed for load balancing in a reverse proxy situation where the looked up values are server names. Example:

  ```
  ##
  ##  map.txt -- rewriting map
  ##

  static   www1|www2|www3|www4
  dynamic  www5|www6
  ```

```
RewriteMap servers rnd:/path/to/file/map.txt
```

- **Hash File**
  MapType: `dbm[=`*`type`*`]`, MapSource: Unix filesystem path to valid regular file

  Here the source is a binary format DBM file containing the same contents as a *Plain Text* format file, but in a special representation which is optimized for really fast lookups. The *type* can be sdbm, gdbm, ndbm, or db depending on compile-time settings[5]. If the *type* is ommitted, the compile-time default will be chosen. You can create such a file with any DBM tool or with the following Perl script. Be sure to adjust it to create the appropriate type of DBM. The example creates an NDBM file.

```
#!/path/to/bin/perl
##
##  txt2dbm -- convert txt map to dbm format
##

use NDBM_File;
use Fcntl;

($txtmap, $dbmmap) = @ARGV;

open(TXT, "<$txtmap") or die "Couldn't open $txtmap!\n";
tie (%DB, 'NDBM_File', $dbmmap,O_RDWR|O_TRUNC|O_CREAT, 0644)
  or die "Couldn't create $dbmmap!\n";

while (<TXT>) {
  next if (/^\s*#/ or /^\s*$/);
  $DB{$1} = $2 if (/^\s*(\S+)\s+(\S+)/);
}

untie %DB;
close(TXT);
```

```
$ txt2dbm map.txt map.db
```

- **Internal Function**
  MapType: `int`, MapSource: Internal Apache function

  Here the source is an internal Apache function. Currently you cannot create your own, but the following functions already exists:

  - **toupper**:
    Converts the looked up key to all upper case.
  - **tolower**:
    Converts the looked up key to all lower case.
  - **escape**:
    Translates special characters in the looked up key to hex-encodings.
  - **unescape**:
    Translates hex-encodings in the looked up key back to special characters.

- **External Rewriting Program**
  MapType: `prg`, MapSource: Unix filesystem path to valid regular file

Here the source is a program, not a map file. To create it you can use the language of your choice, but the result has to be a executable (*i.e.*, either object-code or a script with the magic cookie trick '`#!/path/to/interpreter`' as the first line).

This program is started once at startup of the Apache servers and then communicates with the rewriting engine over its `stdin` and `stdout` file-handles. For each map-function lookup it will receive the key to lookup as a newline-terminated string on `stdin`. It then has to give back the looked-up value as a newline-terminated string on `stdout` or the four-character string ``` ``NULL'' ``` if it fails (*i.e.*, there is no corresponding value for the given key). A trivial program which will implement a 1:1 map (*i.e.*, key == value) could be:

```
#!/usr/bin/perl
$| = 1;
while (<STDIN>) {
    # ...put here any transformations or lookups...
    print $_;
}
```

But be very careful:

1. ``*Keep it simple, stupid*'' (KISS), because if this program hangs it will hang the Apache server when the rule occurs.
2. Avoid one common mistake: never do buffered I/O on `stdout`! This will cause a deadloop! Hence the ``` ``$|=1'' ``` in the above example...
3. Use the `RewriteLock` directive to define a lockfile mod_rewrite can use to synchronize the communication to the program. By default no such synchronization takes place.

The `RewriteMap` directive can occur more than once. For each mapping-function use one `RewriteMap` directive to declare its rewriting mapfile. While you cannot **declare** a map in per-directory context it is of course possible to **use** this map in per-directory context.

Note

For plain text and DBM format files the looked-up keys are cached in-core until the `mtime` of the mapfile changes or the server does a restart. This way you can have map-functions in rules which are used for **every** request. This is no problem, because the external lookup only happens once!

RewriteOptions Directive

Description:	Sets some special options for the rewrite engine
Syntax:	`RewriteOptions` *`Options`*
Default:	`None`
Context:	server config, virtual host, directory, .htaccess
Override:	FileInfo
Status:	Extension
Module:	mod_rewrite

The `RewriteOptions` directive sets some special options for the current per-server or per-directory configuration. The *Option* strings can be one of the following:

- '**inherit**'
  This forces the current configuration to inherit the configuration of the parent. In per-virtual-server context this means that the maps, conditions and rules of the main server are inherited. In per-directory context this means that conditions and rules of the parent directory's `.htaccess` configuration are inherited.

RewriteRule Directive

Description:	Defines rules for the rewriting engine
Syntax:	`RewriteRule` *`Pattern Substitution`*
Default:	`None`
Context:	server config, virtual host, directory, .htaccess
Override:	FileInfo
Status:	Extension
Module:	mod_rewrite
Compatibility:	The cookie-flag is available in Apache 2.0.40 and later.

The `RewriteRule` directive is the real rewriting workhorse. The directive can occur more than once. Each directive then defines one single rewriting rule. The **definition order** of these rules is **important**, because this order is used when applying the rules at run-time.

Pattern is a perl compatible regular expression which gets applied to the current URL. Here ``current'' means the value of the URL when this rule gets applied. This may not be the originally requested URL, because any number of rules may already have matched and made alterations to it.

Some hints about the syntax of regular expressions:

```
Text:
  .           Any single character
  [chars]     Character class: One  of chars
  [^chars]    Character class: None of chars
  text1|text2 Alternative: text1 or text2

Quantifiers:
  ?           0 or 1 of the preceding text
  *           0 or N of the preceding text (N > 0)
  +           1 or N of the preceding text (N > 1)

Grouping:
  (text)      Grouping of text
              (either to set the borders of an alternative or
              for making backreferences where the Nth group can
              be used on the RHS of a RewriteRule with $N)

Anchors:
  ^           Start of line anchor
  $           End   of line anchor

Escaping:
  \char       escape that particular char
              (for instance to specify the chars ".[]()" etc.)
```

For more information about regular expressions have a look at the perl regular expression manpage ("perldoc perlre[6]"). If you are interested in more detailed information about regular expressions and their variants (POSIX regex *etc.*) have a look at the following dedicated book on this topic:

Mastering Regular Expressions
Jeffrey E.F. Friedl
Nutshell Handbook Series
O'Reilly & Associates, Inc. 1997
ISBN 1-56592-257-3

Additionally in mod_rewrite the NOT character ('`!`') is a possible pattern prefix. This gives you the ability to negate a pattern; to say, for instance: ``*if the current URL does **NOT** match this pattern*''. This can be used for exceptional cases, where it is easier to match the negative pattern, or as a last default rule.

Notice

When using the NOT character to negate a pattern you cannot have grouped wildcard parts in the pattern. This is impossible because when the pattern does NOT match, there are no contents for the groups. In consequence, if negated patterns are used, you cannot use `$N` in the substitution string!

Substitution of a rewriting rule is the string which is substituted for (or replaces) the original URL for which *Pattern* matched. Beside plain text you can use

1. back-references `$N` to the RewriteRule pattern
2. back-references `%N` to the last matched RewriteCond pattern
3. server-variables as in rule condition test-strings (`%{VARNAME}`)
4. mapping-function calls (`${mapname:key|default}`)

Back-references are `$`**N** (**N**=0..9) identifiers which will be replaced by the contents of the **N**th group of the matched *Pattern*. The server-variables are the same as for the *TestString* of a `RewriteCond` directive. The mapping-functions come from the `RewriteMap` directive and are explained there. These three types of variables are expanded in the order of the above list.

As already mentioned above, all the rewriting rules are applied to the *Substitution* (in the order of definition in the config file). The URL is **completely replaced** by the *Substitution* and the rewriting process goes on until there are no more rules unless explicitly terminated by a **`L`** flag - see below.

There is a special substitution string named '`-`' which means: **NO substitution**! Sounds silly? No, it is useful to provide rewriting rules which **only** match some URLs but do no substitution, *e.g.*, in conjunction with the **C** (chain) flag to be able to have more than one pattern to be applied before a substitution occurs.

One more note: You can even create URLs in the substitution string containing a query string part. Just use a question mark inside the substitution string to indicate that the following stuff should be re-injected into the QUERY_STRING. When you want to erase an existing query string, end the substitution string with just the question mark.

Note

There is a special feature: When you prefix a substitution field with `http://`*thishost*[*:thisport*] then **mod_rewrite** automatically strips it out. This auto-reduction on implicit external redirect URLs is a useful and important feature when used in combination with a mapping-function which generates the hostname part. Have a look at the first example in the example section below to understand this.

Remember

An unconditional external redirect to your own server will not work with the prefix `http://thishost` because of this feature. To achieve such a self-redirect, you have to use the **R**-flag (see below).

Additionally you can set special flags for *Substitution* by appending

[*flags*]

as the third argument to the `RewriteRule` directive. *Flags* is a comma-separated list of the following flags:

- '**`redirect|R`** **[=*code*]**' (force **r**edirect)
  Prefix *Substitution* with `http://thishost[:thisport]/` (which makes the new URL a URI) to force a external redirection. If no *code* is given a HTTP response of 302 (MOVED TEMPORARILY) is used. If you want to use other response codes in the range 300-400 just specify them as a number or use one of the following symbolic names: `temp` (default), `permanent`, `seeother`. Use it for rules which should canonicalize the URL and give it back to the client, *e.g.*, translate ``` ``/~'' ``` into ``` ``/u/'' ``` or always append a slash to `/u/`*user*, etc.
  Note: When you use this flag, make sure that the substitution field is a valid URL! If not, you are redirecting to an invalid location! And remember that this flag itself only prefixes the URL with `http://thishost[:thisport]/`, rewriting continues. Usually you also want to stop and do the redirection immediately. To stop the rewriting you also have to provide the 'L' flag.
- '**`forbidden|F`**' (force URL to be **f**orbidden)
  This forces the current URL to be forbidden, *i.e.*, it immediately sends back a HTTP response of 403 (FORBIDDEN). Use this flag in conjunction with appropriate RewriteConds to conditionally block some URLs.
- '**`gone|G`**' (force URL to be **g**one)
  This forces the current URL to be gone, *i.e.*, it immediately sends back a HTTP response of 410 (GONE). Use this flag to mark pages which no longer exist as gone.
- '**`proxy|P`**' (force **p**roxy)
  This flag forces the substitution part to be internally forced as a proxy request and immediately (*i.e.*, rewriting rule processing stops here) put through the proxy module[7]. You have to make sure that the substitution string is a valid URI (*e.g.*, typically starting with `http://`*hostname*) which can be handled by the Apache proxy module. If not you get an error from the proxy module. Use this flag to achieve a more powerful implementation of the ProxyPass[8] directive, to map some remote stuff into the namespace of the local server.

  Notice: To use this functionality make sure you have the proxy module compiled into your Apache server program. If you don't know please check whether `mod_proxy.c` is part of the ``` ``httpd -l'' ``` output. If yes, this functionality is available to mod_rewrite. If not, then you first have to rebuild the ``` ``httpd'' ``` program with mod_proxy enabled.
- '**`last|L`**' (**l**ast rule)
  Stop the rewriting process here and don't apply any more rewriting rules. This corresponds to the Perl `last` command or the `break` command from the C language. Use this flag to prevent the currently rewritten URL from being rewritten further by following rules. For example, use it to rewrite the root-path URL ('`/`') to a real one, *e.g.*, '`/e/www/`'.

- '**`next|N`**' (**n**ext round)
  Re-run the rewriting process (starting again with the first rewriting rule). Here the URL to match is again not the original URL but the URL from the last rewriting rule. This corresponds to the Perl `next` command or the `continue` command from the C language. Use this flag to restart the rewriting process, *i.e.*, to immediately go to the top of the loop.
  But be careful not to create an infinite loop!
- '**`chain|C`**' (**c**hained with next rule)
  This flag chains the current rule with the next rule (which itself can be chained with the following rule, *etc.*). This has the following effect: if a rule matches, then processing continues as usual, *i.e.*, the flag has no effect. If the rule does **not** match, then all following chained rules are skipped. For instance, use it to remove the ``` ``.www ```" part inside a per-directory rule set when you let an external redirect happen (where the ``` ``.www ```" part should not to occur!).
- '**`type|T`**=*MIME-type*' (force MIME **t**ype)
  Force the MIME-type of the target file to be *MIME-type*. For instance, this can be used to simulate the `mod_alias` directive `ScriptAlias` which internally forces all files inside the mapped directory to have a MIME type of ``` ``application/x-httpd-cgi ```".
- '**`nosubreq|NS`**' (used only if **n**o internal **s**ub-request)
  This flag forces the rewriting engine to skip a rewriting rule if the current request is an internal sub-request. For instance, sub-requests occur internally in Apache when `mod_include` tries to find out information about possible directory default files (`index.xxx`). On sub-requests it is not always useful and even sometimes causes a failure to if the complete set of rules are applied. Use this flag to exclude some rules.
  Use the following rule for your decision: whenever you prefix some URLs with CGI-scripts to force them to be processed by the CGI-script, the chance is high that you will run into problems (or even overhead) on sub-requests. In these cases, use this flag.
- '**`nocase|NC`**' (**n**o **c**ase)
  This makes the *Pattern* case-insensitive, *i.e.*, there is no difference between 'A-Z' and 'a-z' when *Pattern* is matched against the current URL.
- '**`qsappend|QSA`**' (**q**uery **s**tring **a**ppend)
  This flag forces the rewriting engine to append a query string part in the substitution string to the existing one instead of replacing it. Use this when you want to add more data to the query string via a rewrite rule.
- '**`noescape|NE`**' (**n**o URI **e**scaping of output)
  This flag keeps mod_rewrite from applying the usual URI escaping rules to the result of a rewrite. Ordinarily, special characters (such as '%', '$', ';', and so on) will be escaped into their hexcode equivalents ('%25', '%24', and '%3B', respectively); this flag prevents this from being done. This allows percent symbols to appear in the output, as in

  ```
  RewriteRule /foo/(.*) /bar?arg=P1\%3d$1 [R,NE]
  ```

  which would turn '`/foo/zed`' into a safe request for '`/bar?arg=P1=zed`'.
- '**`passthrough|PT`**' (**p**ass **t**hrough to next handler)
  This flag forces the rewriting engine to set the `uri` field of the internal `request_rec` structure to the value of the `filename` field. This flag is just a hack to be able to post-process the output of `RewriteRule` directives by `Alias`, `ScriptAlias`, `Redirect`, *etc.* directives from other URI-to-filename translators. A trivial example to show the semantics: If you want to rewrite `/abc` to `/def` via the rewriting engine of `mod_rewrite` and then `/def` to `/ghi` with `mod_alias`:

  ```
  RewriteRule ^/abc(.*) /def$1 [PT]
  Alias /def /ghi
  ```

If you omit the `PT` flag then `mod_rewrite` will do its job fine, *i.e.*, it rewrites `uri=/abc/...` to `filename=/def/...` as a full API-compliant URI-to-filename translator should do. Then `mod_alias` comes and tries to do a URI-to-filename transition which will not work.

Note: **You have to use this flag if you want to intermix directives of different modules which contain URL-to-filename translators**. The typical example is the use of `mod_alias` and `mod_rewrite`..

> **For Apache hackers**
>
> If the current Apache API had a filename-to-filename hook additionally to the URI-to-filename hook then we wouldn't need this flag! But without such a hook this flag is the only solution. The Apache Group has discussed this problem and will add such a hook in Apache version 2.0.

- '**`skip|S`**=*num*' (**s**kip next rule(s))
  This flag forces the rewriting engine to skip the next *num* rules in sequence when the current rule matches. Use this to make pseudo if-then-else constructs: The last rule of the then-clause becomes `skip=N` where N is the number of rules in the else-clause. (This is **not** the same as the 'chain|C' flag!)
- '**`env|E=`***VAR*:*VAL*' (set **e**nvironment variable)
  This forces an environment variable named *VAR* to be set to the value *VAL*, where *VAL* can contain regexp backreferences `$N` and `%N` which will be expanded. You can use this flag more than once to set more than one variable. The variables can be later dereferenced in many situations, but usually from within XSSI (via `<!--#echo var="VAR"-->`) or CGI (*e.g.* `$ENV{'VAR'}`). Additionally you can dereference it in a following RewriteCond pattern via `%{ENV:VAR}`. Use this to strip but remember information from URLs.
- '**`cookie|CO=`***NAME*:*VAL*:*domain*[:*lifetime*[:*path*]]' (set **co**cookie)
  This sets a cookie on the client's browser. The cookie's name is specified by *NAME* and the value is *VAL*. The *domain* field is the domain of the cookie, such as '.apache.org',the optional *lifetime* is the lifetime of the cookie in minutes, and the optional *path* is the path of the cookie

> **Note**
>
> Never forget that *Pattern* is applied to a complete URL in per-server configuration files. **But in per-directory configuration files, the per-directory prefix (which always is the same for a specific directory!) is automatically *removed* for the pattern matching and automatically *added* after the substitution has been done.** This feature is essential for many sorts of rewriting, because without this prefix stripping you have to match the parent directory which is not always possible.
>
> There is one exception: If a substitution string starts with ``` ``http://'' ``` then the directory prefix will **not** be added and an external redirect or proxy throughput (if flag **P** is used!) is forced!

> **Note**
>
> To enable the rewriting engine for per-directory configuration files you need to set ``` ``RewriteEngine On'' ``` in these files **and** ``` ``Options FollowSymLinks'' ``` must be enabled. If your administrator has disabled override of `FollowSymLinks` for a user's directory, then you cannot use the rewriting engine. This restriction is needed for security reasons.

Here are all possible substitution combinations and their meanings:

Inside per-server configuration (`httpd.conf`)

for request ``GET /somepath/pathinfo'':

```
Given Rule                                      Resulting Substitution
----------------------------------------------  ----------------------------------
^/somepath(.*) otherpath$1                      not supported, because invalid!

^/somepath(.*) otherpath$1  [R]                 not supported, because invalid!

^/somepath(.*) otherpath$1  [P]                 not supported, because invalid!
----------------------------------------------  ----------------------------------
^/somepath(.*) /otherpath$1                     /otherpath/pathinfo

^/somepath(.*) /otherpath$1 [R]                 http://thishost/otherpath/pathinfo
                                                via external redirection

^/somepath(.*) /otherpath$1 [P]                 not supported, because silly!
----------------------------------------------  ----------------------------------
^/somepath(.*) http://thishost/otherpath$1      /otherpath/pathinfo

^/somepath(.*) http://thishost/otherpath$1 [R]  http://thishost/otherpath/pathinfo
                                                via external redirection

^/somepath(.*) http://thishost/otherpath$1 [P]  not supported, because silly!
----------------------------------------------  ----------------------------------
^/somepath(.*) http://otherhost/otherpath$1     http://otherhost/otherpath/pathinfo
                                                via external redirection

^/somepath(.*) http://otherhost/otherpath$1 [R] http://otherhost/otherpath/pathinfo
                                                via external redirection
                                                (the [R] flag is redundant)

^/somepath(.*) http://otherhost/otherpath$1 [P] http://otherhost/otherpath/pathinfo
                                                via internal proxy
```

Inside per-directory configuration for `/somepath`
(*i.e.*, file `.htaccess` in dir `/physical/path/to/somepath` containing `RewriteBase /somepath`)
for request ``GET /somepath/localpath/pathinfo'':

```
Given Rule                                      Resulting Substitution
----------------------------------------------  ----------------------------------
^localpath(.*) otherpath$1                      /somepath/otherpath/pathinfo

^localpath(.*) otherpath$1  [R]                 http://thishost/somepath/otherpath/pat
                                                via external redirection

^localpath(.*) otherpath$1  [P]                 not supported, because silly!
----------------------------------------------  ----------------------------------
^localpath(.*) /otherpath$1                     /otherpath/pathinfo

^localpath(.*) /otherpath$1 [R]                 http://thishost/otherpath/pathinfo
                                                via external redirection

^localpath(.*) /otherpath$1 [P]                 not supported, because silly!
----------------------------------------------  ----------------------------------
^localpath(.*) http://thishost/otherpath$1      /otherpath/pathinfo

^localpath(.*) http://thishost/otherpath$1 [R]  http://thishost/otherpath/pathinfo
```

```
                                                via external redirection

^localpath(.*) http://thishost/otherpath$1 [P]  not supported, because silly!
----------------------------------------------  ----------------------------------
^localpath(.*) http://otherhost/otherpath$1     http://otherhost/otherpath/pathinfo
                                                via external redirection

^localpath(.*) http://otherhost/otherpath$1 [R] http://otherhost/otherpath/pathinfo
                                                via external redirection
                                                (the [R] flag is redundant)

^localpath(.*) http://otherhost/otherpath$1 [P] http://otherhost/otherpath/pathinfo
                                                via internal proxy
```

Example:

We want to rewrite URLs of the form

`/` *Language* `/~` *Realname* `/.../` *File*

into

`/u/` *Username* `/.../` *File* `.` *Language*

We take the rewrite mapfile from above and save it under `/path/to/file/map.txt`. Then we only have to add the following lines to the Apache server configuration file:

```
RewriteLog   /path/to/file/rewrite.log
RewriteMap   real-to-user               txt:/path/to/file/map.txt
RewriteRule  ^/([^/]+)/~([^/]+)/(.*)$   /u/${real-to-user:$2|nobody}/$3.$1
```

URI References

[1] http://www.engelschall.com/
[3] http://httpd.apache.org/docs-2.1/misc/rewriteguide.html
[4] http://httpd.apache.org/docs-2.1/misc/security_tips.html
[5] http://httpd.apache.org/docs-2.1/install.html#dbm
[6] http://www.perldoc.com/perl5.6.1/pod/perlre.html
[7] http://httpd.apache.org/docs-2.1/mod/mod_proxy.html
[8] http://httpd.apache.org/docs-2.1/mod/mod_proxy.html#proxypass