
mod_rewrite And Friends

Release 3.9.0

Rich Bowen

May 16, 2026

CONTENTS

1	Preface	1
1.1	Who This Book Is For	1
1.2	How To Read This Book	2
1.3	A Note on Versions	2
1.4	Errata and Feedback	2
1.5	Acknowledgments	2
2	Regular Expressions	3
2.1	The Building Blocks	3
2.2	Matching anything	5
2.3	Escaping characters	5
2.4	Anchoring text	6
2.5	Matching one or more characters	7
2.6	Matching zero or more characters	7
2.7	Repetition quantifiers	7
2.8	Greedy Matching	8
2.9	Making a match optional	8
2.10	Grouping and capturing	9
2.11	Backreferences	9
2.12	Character Classes	10
2.13	Negation	11
2.14	Regex examples	11
2.14.1	Email address	11
2.14.2	Phone number	12
2.14.3	Matching URIs	13
2.14.4	Matching the homepage	13
2.14.5	Matching a directory	14
2.14.6	Matching a filetype	14
2.15	Regex tools	14
2.15.1	Online tools	14
2.15.2	Command-line tools	15
2.15.3	Learning resources	16
2.16	RewriteRule generators	16
2.17	Summary	16

3	URL Mapping	17
3.1	mod_rewrite	17
3.2	DocumentRoot	17
3.3	DirectoryIndex	18
3.4	FallbackResource	18
3.5	Automatic directory listings	19
3.6	Alias	20
3.7	AliasMatch and ScriptAliasMatch	20
3.8	Redirect	21
3.9	RedirectMatch	21
3.10	Location	22
3.10.1	Virtual Hosts	23
3.11	mod_vhost_alias	23
3.11.1	Proxying	24
3.12	mod_proxy_express	24
3.12.1	mod_actions	24
3.12.2	mod_imagemap	25
3.12.3	mod_negotiation	25
3.12.4	mod_userdir	26
3.12.5	mod_speling	26
3.12.6	File not found	27
4	Introduction to mod_rewrite	29
4.1	Where mod_rewrite fits in the request lifecycle	29
4.1.1	Loading mod_rewrite	29
4.1.2	RewriteEngine	31
4.2	How mod_rewrite interacts with other modules	32
4.2.1	mod_rewrite in .htaccess files	32
4.3	What are .htaccess files?	32
4.4	Ok, so, what's the deal with mod_rewrite in .htaccess files?	34
4.5	So, what do I do?	35
4.5.1	RewriteOptions	35
4.6	Inherit and InheritBefore	35
4.7	InheritDown and InheritDownBefore	36
4.8	IgnoreInherit	36
4.9	AllowNoSlash	36
4.10	AllowAnyURI	36
4.11	MergeBase	36
4.12	IgnoreContextInfo	37
4.13	LegacyPrefixDocRoot	37
4.14	LongURLOptimization	37
4.14.1	RewriteBase	37
5	RewriteRule	39
5.1	Syntax	39
5.2	Pattern	39
5.2.1	Negated patterns	40
5.3	Target	40

5.3.1	A file-system path	40
5.3.2	URL-path	41
5.3.3	Absolute URL	41
5.3.4	- (dash)	41
5.4	Backreferences	41
5.4.1	Server variables in the target	42
5.5	Query string handling	42
5.6	Per-directory context gotchas	43
5.7	Home directory expansion	44
5.8	How rules are processed	44
5.9	Flags at a glance	44
5.10	Security Considerations	45
5.10.1	Open Redirects	45
5.10.2	Server-Side Request Forgery (SSRF)	46
5.10.3	Path Traversal	46
5.10.4	General Principles	46
6	Rewrite Logging	47
6.1	Trace Levels	47
6.2	Enabling Rewrite Logging	48
6.3	Per-Directory Logging	48
6.4	What's in the Rewrite Log? — An Example	49
6.5	RewriteRules in .htaccess Files — An Example	52
6.6	Debugging RewriteMap Lookups	53
6.7	Common Error Messages	53
6.8	Don't Leave It On	54
7	RewriteRule Flags	55
7.1	B - escape backreferences	55
7.2	BNP - backrefnoplus	56
7.3	BCTLS	56
7.4	BNE	57
7.5	C - chain	57
7.6	CO - cookie	57
7.6.1	Domain	57
7.6.2	Lifetime	57
7.6.3	Path	58
7.6.4	Secure	58
7.6.5	httponly	58
7.6.6	Example	58
7.7	DPI - discardpath	58
7.8	E - env	59
7.9	END	60
7.10	F - forbidden	60
7.11	G - gone	61
7.12	H - handler	61
7.13	L - last	61
7.14	N - next	63

7.15	NC - nocase	63
7.16	NE - noescape	63
7.17	NS - nosubreq	64
7.18	P - proxy	64
7.19	PT - passthrough	65
7.20	QSA - qsappend	65
7.21	QSD - qsdiscard	65
7.22	QSL - qslast	66
7.23	R - redirect	66
7.24	S - skip	66
7.25	T - type	67
7.26	UnsafeAllow3F	68
7.27	UnsafePrefixStat	68
7.28	UNC	69
8	RewriteCond	71
8.1	TestString	71
8.2	CondPattern	74
8.3	Examples	79
8.3.1	Matching query strings	79
8.3.2	Hostname-based routing	79
8.3.3	File and directory existence	80
8.3.4	Time-based rules	80
8.3.5	Requiring HTTPS	80
8.3.6	Combining conditions with OR	81
9	RewriteMap	83
9.1	Creating a RewriteMap	83
9.2	Using a RewriteMap	84
9.3	Default Values	84
9.4	RewriteMap Types	84
9.4.1	int	84
9.4.2	toupper	84
9.4.3	tolower	85
9.4.4	escape	85
9.4.5	unescape	85
9.4.6	txt	85
9.4.7	rnd	86
9.4.8	dbm	87
9.4.9	prg	87
9.4.10	dbd	88
10	Proxies and mod_rewrite	91
10.1	The mod_proxy family	91
10.1.1	Which modules to load	93
10.2	When to use [P] vs ProxyPass	93
10.3	Basic proxying with [P]	94
10.4	Conditional proxying	94

10.5	Proxying with RewriteMap	95
10.6	SSL/TLS considerations	95
10.7	Rewriting proxied responses	96
10.8	Common pitfalls	96
11	Virtual hosts and mod_rewrite	97
11.1	The problem	97
11.2	Dynamic vhosts with mod_rewrite	98
11.2.1	Stripping the www. prefix	98
11.3	Using a map file for vhosts	98
11.4	Handling aliases and CGI in dynamic vhosts	99
11.5	Why mod_vhost_alias is usually better	100
11.6	Per-user virtual hosts	100
11.7	Logging for dynamic vhosts	101
12	Access control with mod_rewrite	103
12.1	Forbidding image hotlinking	103
12.1.1	The non-rewrite alternative	104
12.2	Blocking specific robots/user agents	104
12.2.1	The non-rewrite alternative	105
12.3	Denying by IP address or hostname	105
12.3.1	The modern alternative	106
12.4	Referer-based deflection	106
12.5	Time-based access control	107
12.5.1	The <If> alternative	107
12.6	Requiring HTTPS	107
12.7	Environment variable gating	108
12.8	When not to use mod_rewrite for access control	109
13	Conditional Configuration	111
13.1	Introduction	111
13.2	Match Directives	111
13.3	IfDefine	112
13.4	Define	113
13.5	<If>, <Elseif>, and <Else>	114
13.5.1	Canonical hostname	114
13.5.2	Image hotlinking	114
13.5.3	The expression parser (ap_expr)	115
13.6	mod_macro	115
13.7	mod_proxy_express	116
13.8	mod_vhost_alias	116
13.9	Conditional logging	117
13.9.1	env=	117
13.9.2	Response-code conditional format strings	119
13.9.3	Per-module logging	119
13.9.4	Per-directory logging	119
13.9.5	Piped logging	120

14	Content Munging	121
14.1	mod_substitute	121
14.1.1	Basic usage	121
14.1.2	Handling long lines	122
14.1.3	Using regular expressions	123
14.1.4	Content type filtering	123
14.2	mod_sed	123
14.2.1	Basic usage	123
14.2.2	When to reach for mod_sed	124
14.3	mod_proxy_html	124
14.3.1	A worked example	124
14.3.2	Extending to CSS and JavaScript	125
14.3.3	Setting the output doctype	125
14.3.4	Comparison with mod_substitute	125
14.4	Filters	126
14.4.1	Adding filters to the chain	126
14.4.2	Filter ordering	126
14.4.3	Conditional filtering with mod_filter	126
14.4.4	The escape hatch: mod_ext_filter	127
14.4.5	Putting it all together	127
15	Recipes	129
15.1	Common Redirects	129
15.1.1	Redirecting HTTP to HTTPS	129
15.1.2	Canonicalizing the Hostname (www vs. non-www)	131
15.1.3	Adding or Removing Trailing Slashes	132
15.1.4	Redirecting an Entire Site to a New Domain	133
15.1.5	Redirecting Individual Pages That Have Moved	134
15.1.6	Redirecting Wildcard Subdomains	135
15.2	Clean and Pretty URLs	136
15.2.1	Removing File Extensions (.php, .html)	137
15.2.2	Front Controller Pattern (CMS/Framework Routing)	138
15.2.3	Mapping Clean URL Paths to Query Parameters	140
15.3	Access Control	141
15.3.1	Blocking Hotlinking (Referrer-Based Access)	141
15.3.2	Blocking Requests by User-Agent	142
15.3.3	Cookie-Based Redirect to Login Page	143
15.3.4	IP-Based Access Control	144
15.4	Proxying	145
15.4.1	Rewriting URLs for a Reverse Proxy Backend	145
15.4.2	Redirects Behind a TLS-Terminating Proxy	147
15.4.3	WebSocket Proxying	148
15.5	Query String Manipulation	149
15.5.1	Capturing and Rewriting Query Strings	149
15.5.2	Stripping Query Strings	150
15.5.3	Using SetEnvIf with Query Strings	151
15.6	Edge Cases and Gotchas	152
15.6.1	Diagnosing and Fixing Rewrite Loops	152

15.6.2	.htaccess vs. Server Config Context Differences	154
15.6.3	Rule Ordering and the [L] Flag	155
15.6.4	Debugging Rewrite Rules with the Rewrite Log	157
15.6.5	Serving a Fallback Resource When a File Is Missing	158
15.6.6	Maintenance Mode (503 Service Unavailable)	159
15.6.7	Handling Special Characters and Encoded URLs	161
15.6.8	Performance with Large Numbers of Redirects	162
15.7	When NOT to Use mod_rewrite	163
15.7.1	Simple Redirects: Use Redirect, Not RewriteRule	163
15.7.2	Proxying: Use ProxyPass, Not RewriteRule [P]	164
15.7.3	Conditional Logic: Use <If> Expressions	165
15.7.4	Fallback Resources: Use FallbackResource, Not RewriteRule	166
15.7.5	Advanced: Using RewriteMap for Dynamic Rewrites	167
15.7.6	Advanced: IP Range Matching with RewriteMap	169
16	Glossary	171
17	List of Epigraphs	175
17.1	Preface	175
17.2	Chapter 1: Regular Expressions	175
17.3	Chapter 2: URL Mapping	176
17.4	Chapter 3: Introduction to mod_rewrite	176
17.5	Chapter 4: RewriteRule	176
17.6	Chapter 5: Rewrite Logging	177
17.7	Chapter 6: RewriteRule Flags	177
17.8	Chapter 7: RewriteCond	177
17.9	Chapter 8: RewriteMap	178
17.10	Chapter 9: Proxies and mod_rewrite	178
17.11	Chapter 10: Virtual hosts and mod_rewrite	178
17.12	Chapter 11: Access control with mod_rewrite	178
17.13	Chapter 12: Conditional Configuration	179
17.14	Chapter 13: Content Munging	179
17.15	Chapter 14: Recipes	180
Index		181

PREFACE

They stared at the branch. There wasn't just one flower out there, there were dozens, although the frogs weren't able to think like this because frogs can't count beyond one.

They saw lots of ones.

—Terry Pratchett, *Wings*

`mod_rewrite` is the Swiss Army knife of the Apache HTTP Server. It is also, by a wide margin, the most misunderstood module in the entire server distribution. In my years of answering questions on the httpd support channels, I've observed that most of the confusion around `mod_rewrite` comes not from the module itself, but from the regular expressions it relies on, and from a lack of awareness that simpler tools often exist to do the job.

This book has two goals. The first is to teach you `mod_rewrite` — how it works, how to read and write rewrite rules, how to debug them when they go wrong, and how to know when you've got the right solution. The second, and perhaps more important, goal is to teach you when *not* to use `mod_rewrite`. The Apache HTTP Server ships with a remarkable number of URL mapping tools, and reaching for `mod_rewrite` first is a common and unnecessary habit.

The title — *mod_rewrite And Friends* — reflects this. We will spend considerable time on `mod_rewrite`, but also on `mod_alias`, `mod_proxy`, `<If>` blocks, and the various other modules and directives that handle URL mapping, content negotiation, and request routing. Understanding all of these tools means you'll choose the right one for the job, rather than hammering everything with the same regular expression-shaped nail.

1.1 Who This Book Is For

This book is for anyone who administers or develops for the Apache HTTP Server and needs to control how URLs are handled. That includes system administrators managing virtual hosts, web developers wrangling redirects, and the occasional desperate soul staring at a `.htaccess` file at 2 AM wondering why nothing works.

I assume you're comfortable with a text editor and a terminal. I do *not* assume you already know regular expressions — *Chapter 1* covers them from scratch.

1.2 How To Read This Book

The chapters are arranged to build on one another, starting with regular expressions (Chapter 1) and URL mapping fundamentals (Chapter 2) before moving into `mod_rewrite` specifics. If you already know regex and just want the `mod_rewrite` details, skip ahead to Chapter 3.

Chapters 3 through 8 form the core reference: the `RewriteRule` directive, logging, flags, `RewriteCond`, and `RewriteMap`. These build on each other, so reading them in order is worthwhile.

Chapters 9 through 13 are topical — proxying, virtual hosts, access control, conditional configuration, and content manipulation. Each one stands alone, so read whichever is relevant to your current problem.

Chapter 14 gathers common recipes in one place for quick reference. Each chapter includes practical examples, because `mod_rewrite` is understood far more quickly through examples than through lectures.

1.3 A Note on Versions

The examples in this book target Apache HTTP Server 2.4 and later. If you're running something older than 2.4, you should strongly consider upgrading — not just for `mod_rewrite`, but for the significant improvements in configuration flexibility, security, and performance.

1.4 Errata and Feedback

Found an error, have a suggestion, or want to contribute? Please file an issue at https://github.com/rbowen/mod_rewrite_book/issues.

1.5 Acknowledgments

This book has been a work in progress since 2013, and I expect it to remain in progress for some time to come. But I think it's time to call it . . . if not done, then ready to see the light.

Thanks are owed to the Apache HTTP Server documentation team, whose work I have drawn on extensively, and to the many people on the `#httpd` IRC channel (and later, Slack channel) and the `httpd` mailing lists who have, over the years, asked the questions that shaped the content of this book.

And to my muses, Rhi, Z, and E: I love you more than the whole world.

Finally, to Maria, who makes everything beautiful. And so that was all right, Best Beloved. Do you see?

– Rich Bowen, April 2026

Trademarks

Apache, Apache HTTP Server, and the Apache oak leaf logo are trademarks of The Apache Software Foundation. All other trademarks are the property of their respective owners. Use of these trademarks does not imply endorsement by The Apache Software Foundation. This book is not affiliated with or endorsed by The Apache Software Foundation.

REGULAR EXPRESSIONS

The first thing you learn in life is you're a fool.
The last thing you learn in life is you're the same fool.

—Ray Bradbury, *Dandelion Wine*

Much of the content in this book requires that you have some mastery of regular expressions. Indeed, in my years of teaching `mod_rewrite`, it has been my observation that most people don't find `mod_rewrite` hard at all: they're just intimidated by regular expressions.

There is one excellent book about regular expressions, and if you want to become a regular expression (or “regex”) guru, you should get it. That book is [Mastering Regular Expressions](#) by Jeffrey Friedl. The third edition dates from 2006, but regular expression fundamentals have not changed significantly since then, and it remains the gold standard on the subject — no other book has come close to replacing it.

Two other books are worth mentioning: [Regular Expressions Cookbook](#) by Jan Goyvaerts and Steven Levithan (O'Reilly, 2nd edition 2012) is a practical, recipe-oriented companion, and the [Regular Expression Pocket Reference](#) by Tony Stubblebine (O'Reilly, 2nd edition 2007) is handy to keep nearby when you need a quick syntax reminder.

Finally, even if you never intend to write a line of Perl, the Perl regular expression man page (`man perlre`) is one of the best regex references available on any Unix system. It's thorough, well-organized, and covers every feature of PCRE — which is the regex flavor that `mod_rewrite` uses. It's already installed on most systems.

If you just want to know enough about regex to master `mod_rewrite`, read this chapter a few times, and that should be sufficient.

The goal of this chapter is to introduce the building blocks - the basic vocabulary - and then discuss some of the arcana of crafting your own regular expressions, as well as reading those that others have bequeathed to you. If you are already reasonably familiar with regex syntax, you can safely skip this chapter.

2.1 The Building Blocks

Regular expressions are a means to describe a text pattern,² so that you can look for that pattern in a block of data. The best way to read any regular expression is one character at a time, so you need to know what each character represents.

² Technically, it's any data, but in the context of Apache httpd, we're primarily interested in text as it appears in URLs

These are the basic building blocks that you will use when writing regular expressions. If you don't already know regex syntax, you'll want to stick a bookmark on this page, since you'll be referring to it until you become familiar with these characters. The **Regular Expression Vocabulary** table is your key to translating a line of seemingly random characters into a meaningful pattern. The table will be followed by further explanations and examples for each of the items in the table.

Regular Expression Vocabulary

Char	Meaning
.	Any character
\	Escapes a character that has a special meaning. Thus, \. means a literal . character. You can match a literal \ character by using \\ . Additionally, placing \ in front of a regular character can add a special meaning to that character. For example, \t means a tab character. See <i>Escaping characters</i> for more detail on that.
^	An anchor which insists that the pattern start at the beginning of the string. ^A means that the string must start with A.
\$	An anchor which insists that the string ends with the specified pattern. X\$ means that the string must end with X.
+	Match the previous thing one or more times. So a+ means one or more a's.
*	Match the previous thing zero or more times. This is the same as +, except that it's also acceptable if the thing wasn't there at all.
?	Match the previous thing zero or one times. In other words, it makes it optional. It also makes the * and + characters non-greedy. See <i>Greedy Matching</i> .
{n, m}	Indicates that the previous thing should match at least n, and not more than m times. For example, a{2,7} matches at least 2, and not more than 7, occurrences of the letter a
()	Provides grouping and capturing functions. Grouping means treating more than one character as though they were a single unit. You can apply repetition characters to a group created in this way. Capturing means remembering the thing that matched, so that we can use it again later. This is called a 'backreference.'
[]	Called a "character class," this matches only one of the contained characters. For example, [abc] matches a single character which is either a or b or c.
^	Negates a match within a character set. (Remember that outside of a character class, it means something else. See above.) Thus, [^abc] matches a single character which is neither a nor b nor c.
!	Placed on the front of a regular expression, this means "NOT". That is, it negates the match, and so succeeds only if the string does not match the pattern.

That's not all there is to regular expressions, but it's a really good starting point. Each regular expression presented in this book will have an explanation of what it's doing, which will help you see in practical examples what each of the above characters actually ends up meaning in the wild. And, in my experience, regular expressions are understood much more quickly via examples rather than via lectures.

What follows is a more detailed explanation of each of the items in the table above, with examples.

2.2 Matching anything

The `.` character in a regular expression matches any character. For example, consider the following pattern:

```
a.c
```

That pattern matches a string containing `a`, followed by any character, followed by `c`. So, that pattern matches the strings “abc”, “ancient”, and “warcraft”, each of which contain that pattern. It does not match “tragic”, on the other hand, because there are two characters between the `a` and the `c`. That is, the `.` by itself, matches a single character only.

The `.` character is very frequently used in connection with `*` to mean “match everything”. You’ll see the `(.*)` pattern appearing often throughout this book, and throughout examples that you see online. And while it’s often what you want, it’s just as often used incorrectly. Remember that while `(.*)` matches any string, so will the simpler and faster pattern `^` because every string has a start (even an empty string) and so `^` matches it.

It’s faster, too, because while `(.*)` has to match all the way out to the end of the string, `^` only has to note that the string has a beginning, and then it is done. Note also that the pattern `(.*)` has parentheses and therefore captures the matched string into the variable `$1`. If you’re not planning to use `$1` in a later substitution, then this, in addition to being a waste of computation cycles, is a waste of memory.

While considerations of this kind probably won’t save you a noticeable amount of time, getting into the habit of writing efficient regular expressions will, in the long run, not only save you these small amounts, but will result in rules that are easier to understand and easier to maintain, because they match only what you’re interested in, and nothing more.

2.3 Escaping characters

The backslash, or escape character, either adds special meaning to a character, or removes it, depending on the context. For example, you’ve already been told that the `.` character has special meaning. But if you want to match the literal `.` character, then you need to escape it with the backslash. So, while `.` means “any character,” `\.` means a literal “.” character.

Conversely, some characters gain special meaning when prefixed by a `\` character. For example, while `s` means a literal “s” character, `\s` means a “whitespace” character. That is, a space or a tab.

The **Metacharacter** table below lists useful escape characters that you’ll see throughout the book and can be used as shorthand for more verbose patterns.

Metacharacters

Char-acter	Meaning
d	Match any character in the range 0 - 9
D	Match any character NOT in the range 0 - 9
s	Match any whitespace characters (space, tab etc.).
S	Match any character NOT whitespace (space, tab).
w	Match any character in the range 0 - 9, A - Z and a - z
W	Match any character NOT the range 0 - 9, A - Z and a - z
b	Word boundary. Match any character(s) at the beginning (<code>\babc</code>) and/or end (<code>abc\b</code>) of a word, thus <code>\bcow\b</code> will match cow but not cows, but <code>\bcow</code> will match cows.
B	Not a word boundary. Match any character(s) NOT at the beginning (<code>\Babc</code>) and/or end (<code>cow\B</code>) of a word, thus <code>\Bcow\B</code> will match scows but not cows, but <code>cow\B</code> will match coward.
t	Match a tab character
n	Match a newline character
x	Matches a character with a particular hex code. For example, <code>\x5A</code> would match a Z, which has a hex code of 5A.

The term “metacharacter” is often also applied to characters such as `.` and `$` which have special meanings within regular expressions.

2.4 Anchoring text

Referred to as anchor characters, these ensure that a string starts with, or ends with, a particular character, or sequence of characters. Since this is a very common need, these are included in this basic vocabulary. Consider the examples in the `anchor examples table`

Anchor examples

Ex-ample	Meaning
<code>^/</code>	This matches any string that starts with a slash
<code>.jpg\$</code>	This pattern matches any string that ends with .jpg.
<code>^/\$</code>	Matches a string that starts with, and ends with, a slash. That is, it will only match a string that is a single slash, and nothing else.
<code>^\$</code>	Matches an empty string - that is, a string that has nothing between its start and its end.

Remember, as you craft your regular expressions, that they are, by default, a substring match. Which is to say, a pattern of `cow` matches `cow`, `scow`, `coward`, and `pericowperitis`, because they all contain “cow” somewhere in them. Using the anchor characters allows you to be more specific as to what you wanted to match. The `\b` metacharacter, introduced above, can also be useful in some contexts, but perhaps less so when you’re dealing with URLs.

2.5 Matching one or more characters

The + character allows a pattern or character to match more than once. For example, the following pattern will allow for common misspellings of the word “giraffe”.

```
giraf+e+
```

This pattern will allow one or more f’s, as well as one or more e’s. So it matches “girafe”, “giraffe”, and “giraffee”. It will also match “girafffffeeeee”.

Be sure to use + rather than * when you want to ensure non-empty matches.

2.6 Matching zero or more characters

The * character allows the previous character to match zero or more times. That is to say, it’s exactly the same as +, except that it also allows for the pattern to not match at all. This is often used when + was meant, which can result in some confusion when it matches an empty string. As an example, we’ll use a slight modification of the pattern used in the above section:

```
giraf*e*
```

This pattern matches the same strings listed above (“giraffe”, “girafe” and “giraffee”) but will also match the string “giraeeee”, which contains zero “f” characters, as well as the string “gira”, which contains zero “f” characters and zero “e” characters.

Most commonly, you’ll see it used in conjunction with the . character, meaning “match anything.” Frequently, in that case, the person using it has forgotten that regular expressions are substring matches. For example, consider this pattern:

```
.*\.gif$
```

The intent of that pattern is to match any string ending in .gif. The \$ insists that it is at the end of the string, and the \ before the . makes that a literal . character, rather than the wildcard . character. In this particular case, the .* was there to mean “starts with anything,” but is completely unnecessary, and will only serve to consume time in the matching process.

A more useful example of the * character is one which checks for a comment line in an Apache configuration file. The first non-space character needs to be a #, but the spaces are optional:

```
^\s*#
```

This pattern, then, matches a string that might (but doesn’t have to) begin with whitespace, followed by a #. This ensures that the first non-space character of the line is a #.

2.7 Repetition quantifiers

If you want to match a particular number of times, you can use the {n,m} quantifier to specify the range of times you wish to match. The possibilities of how you can specify this are shown in the table below.

Repetition quantifiers

Pattern	Meaning
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match at least n times, but not more than m times

These repetition quantifiers may be applied to a single character, or to a grouping. For example:

```
\d{1,3}
```

will match 1, 2, or 3 digits.

```
[abc]{2,5}
```

Will match anywhere from 2 to 5 instances of a, b, or c.

2.8 Greedy Matching

In the case of all of the repetition characters above, matching is greedy. That is, the regular expression matches as much as it possibly can. That is, if you apply the regular expression `a+` to the string `aaaa`, matches the entire string, and not be satisfied by just the first `a`. This is particularly important when you are using the `.*` syntax, which can occasionally match more than you thought it would. I'll give some examples of this after we've discussed a few more metacharacters.

On the other hand, if you wish for matches to not be greedy, you can offset the greedy nature of the repetition character by putting a `?` after it.

Consider, for example, a scenario where I want to match everything between two slashes in a URL. I'll be applying the regular expression to the URI `/one/two/three/`, and I'll try a greedy, and not-greedy, regular expression. The [table of greedy examples](#) shows the results of these patterns.

Examples of greedy matching

Greedy?	Pattern	Matches
Yes	<code>/(.*)/</code>	<code>one/two/three</code>
No	<code>/(.*?)/</code>	<code>one</code>

The first regex is greedy, and matches as much as it possibly can, going out to the last slash. The second is non-greedy, and so stops as early as it can, when it encounters the second slash.

2.9 Making a match optional

The `?` character makes a single character match optional. This is extremely useful for common misspellings, or elements that may, or may not, appear in a string. For example, you might use it in a word when you're not sure whether it's supposed to be hyphenated:

```
e-?mail
```

The above pattern matches both “email” and “e-mail”, so that either spelling will be accepted. Likewise, you could use:

```
colou?r
```

to match the word color both as it is spelled in the USA, and the way that it is spelled in the rest of the world. Additionally, the ? character turns off the “greedy” nature of the + and * characters. Thus, putting a ? after a + or a * will make it match as little as it possibly can. See *Greedy Matching*.

Further examples of the greedy vs. non-greedy behavior will follow once we have learned about backreferences.

2.10 Grouping and capturing

Parentheses allow you to group several characters as a unit, and also to capture the results of a match for later use. The ability to treat several characters as a unit is extremely useful in pattern matching. Think of it as combining several atoms into a single molecule. For example, consider this example:

```
(abc)+
```

This will look for the sequence “abc” appearing one or more times, and so would match the string “abc” and the string “abcabc”.

2.11 Backreferences

Even more useful is the “capturing” functionality of the parentheses. Once a pattern has matched, you often want to know what matched, so that you can use it later. This is usually referred to as “backreferences.”

For example, you may be looking for a .gif file, as in the example above, and you really want to know what .gif file you matched. By capturing the filename with parentheses, you can use it later on:

```
(.*\.gif)$
```

In the event that this pattern matches, we will capture the matching value in a special variable, \$1. (In some contexts, the variable may be called %1 instead.) If you have more than one set of parentheses, the second one will be captured to the variable \$2, the third to \$3, and so on. Only values up through \$9 are available, however. The reason for this is that \$10 would be ambiguous. It might mean \$1, followed by a literal zero (0), or it might mean \$10. Rather than providing additional syntax to disambiguate this term, the designer of mod_rewrite instead chose to only provide backreferences through \$9.

The exact way in which you can exploit this feature will become clearer once I start looking at the RewriteRule directive in *RewriteRule*

Consider these two patterns, applied to the string “canadian”.

```
c(.*?)n
c(.*?)n
```

The first pattern will return with a value of “anadia” in \$1, since it will match as much as it possibly can between the first c and the last n it sees. The second, on the other hand, will return with \$1 set to “a”, since it is non-greedy, and so stops at the first n it sees.

It is instructive to use a tool such as [regex101](#) or [pcre2test](#), described in the *Regex tools* section below, and feed them these patterns and strings, to watch them match the different parts of the string. **Mastering Regular Expressions** also has a very complete treatment of backreferences, greedy matching, and what actually happens during the matching phase.

2.12 Character Classes

A character class allows you to define a set of characters, and match any one of them. There are several built-in character classes, like the `\s` metacharacter that you saw above. Using the `[]` notation lets you define your own custom character classes. As a very simple example, consider the following:

```
[abc]
```

This character class matches the letter a, or the letter b, or the letter c. For example, if we wanted to match the subset of users whose usernames started with one of those letters, we might look for the pattern:

```
/home/([abc].*)
```

This combines several of the characters that have been described above. It ends up matching a directory path for that subset of users, and the username ends up in the \$1 variable. Well, actually, not quite, as we’ll see in a minute, but almost.

The character class syntax also allows you to specify a range of characters fairly easily. For example, if you wanted to match a number between 1 and 5, you can use the character class `[1-5]`.

Within a character class, the `^` character has special meaning, if it is the first character in the class. The character class `[^abc]` is the opposite of the character class `[abc]`. That is, it matches any character which is not a, b, or c.

Which brings us back to the example above, where we are attempting to match a username starting with a, b, or c. The problem with the example is that the `*` character is greedy, meaning that it attempts to match as much as it possibly can. If we want to force it to stop matching when it reaches a slash, we need to match only “not slash” characters:

```
/home/([abc][^/]+)
```

I’ve replaced the `.*` with `[^/]+` which has the effect that, rather than matching any character, it matches only not-slash characters. In other words, it will only match up to a slash, or the end of the string, whichever comes first. Also, I’ve used `+` instead of `*`, since one-character usernames are typically not permitted. Now, \$1 will contain the username, whereas, before, it could possibly have contained other directory path components after the username.

2.13 Negation

Finally, if you wish to negate an entire regular expression match, prefix it with `!`. This is not consistent across all regular expression implementations, but can be used in a number of them. A very common use of this in the context of rewrite rules will be to indicate that you want a pattern to apply to all directories except for one. So, for example, if we wanted to exclude the `/images` directory from consideration, we would match the `/images` directory, but then negate the match, thus:

```
!^/images
```

This matches any path not starting with `/images`. We'll see more of this kind of pattern match especially in the chapter *Proxies and mod_rewrite*.

2.14 Regex examples

A few examples may be instructive in your understanding of how regular expressions work. We'll start with a few of the cases that you may frequently encounter, and suggest a few alternate solutions to each.

2.14.1 Email address

We'll start with a common favorite. You want to craft a regular expression that matches an email address. The general format of an email address is “something @ something dot something”. When you are crafting a regular expression from scratch, it's good to express the pattern to yourself in terms like this, because it's a good start towards writing the expression itself.

To express this as a regular expression, let's take the component parts. The catch all “something” part can likely be expressed as `.*` and the `.` and `@` parts are literal characters. So, this gives us a starting point of:

```
.*@.*\..*
```

This is a good start, and matches most email addresses. It will probably match all email addresses. However, it will also match a lot of stuff that isn't email addresses, like “`@@.@`”, “`@.com`”, and “This isn't an `em@il` address.” So we have to try something a little more specific.

We want to require that the “something” before the `@` sign is not zero length, and contains certain types of characters. For example, it should be alpha-numeric, but may also contain certain other special characters, like dot, underscore, or dash.

Fortunately, PCRE provides us with a convenient way to say “alpha-numeric characters”, using a named character class. There are quite a number of these, such as `[:alpha:]` to match letters, `[:digit:]` to match numbers 0 through 9, and `[:alnum:]` to match alpha-numeric characters.

Next, we want to ensure that the domain name part of the pattern is alphanumeric too, except that the top level domain (tld), i.e., the last part of the domain name, must be letters.

And we want to allow an arbitrary number of dots in the hostname, so that “`a.com`” and “`mail.s.ms.uky.edu`” are both valid hostname portions of an email address. So we can say the above description as:

```
^[[[:alnum:]]_.-]+@([[[:alnum:]]+\.)+[[[:alpha:]]]+$
```

This is far more specific, and will match most valid email addresses. However, it will also exclude a few edge-cases, as well as allowing some things that are not valid addresses, such as invalid domain names.

You should note that this was something of a fool's errand - there does not exist a regular expression that matches all possible email addresses. Indeed, I started with this example to give you a flavor for just how complicated it can be to craft a pattern for something that is not well defined.

For more discussion of writing regular expressions to match email addresses, simply search for `email regex` in your favorite search engine, and you'll find many, many articles about how and why it is impossible.

2.14.2 Phone number

Next we'll consider the problem of matching a phone number. This is much harder than it would at first appear. We'll assume, for the sake of simplicity, that we're just trying to match US phone numbers, which are 10 numbers.

The number consists of three numbers, then three more, then four more. These numbers may, or may not, be separated by a variety of things. The first three may or may not be enclosed in parentheses. So we'll try something like this:

```
\(?:\d{3}\)?[-. ]?\d{3}[-. ]?\d{4}
```

This pattern matches most US phone numbers, in most of the ordinary formats. The first three numbers may or may not be in parentheses, and the blocks of numbers may or may not be separated by dashes (-), dots (.) or spaces.

It is still far from foolproof, because users will come up with ways to submit data in unexpected format.

Let's go through the rule one piece at a time:

`\(?:` - This sub-pattern represents an optional opening parenthesis. The backslash is necessary because parentheses already have special meaning in regular expressions. We want to remove that special meaning, and have a literal opening parenthesis. The question mark makes this character optional. That is, the person entering the data may or may not enclose the first three numbers with parentheses, and we want to ensure that either one is acceptable.

`\d{3}` - `\d` means a digit. (Remember: `d` for digit.) This can also be written as `[:digit:]`, but the `\d` notation tends to be more common, for the simple reason that it's less to type. The `{3}` following the `\d` indicates that we want to match the character exactly three times. That is, we require three digits in this portion of the match, or it will return failure.

See the section *Repetition quantifiers* for the various syntaxes you can use to indicate the number of repetitions you want.

`\)?` - Like the opening parenthesis we started with, this is an optional closing parenthesis.

`[-.]?` - Another optional character, this allows, but does not require, a dash, a dot, or a space, to appear between the first three numbers and the next three numbers.

If you discover that your users are separating blocks with some other character, you can add that to the character class. So, for example, if they are using an underscore, you would modify this part of the pattern to be `[-._]` instead, to include this new character.

The rest of the expression is exactly the same as what we have already done, except that the last block of numbers contains 4 numbers, rather than three.

The next step in crafting a regular expression is to think of the ways in which your pattern will break, and whether it is worth the additional work to catch these edge cases. For example, some users will enter a 1 before the entire number. Some phone numbers will have an extension number on the end. And that one hard-to-please user will insist on separating the numbers with a slash rather than one of the characters we have specified. These can probably be solved with a more complex regex, but the increased complexity comes at the price of speed, as well as a loss of readability. It took a page to explain what the current regex does, and that's at least some indication of how much time it would take you to decipher a regex when you come back to it in a few months and have forgotten what it is supposed to be doing.

2.14.3 Matching URIs

Finally, since this is, after all, a book about `mod_rewrite`, it seems reasonable to give some examples of matching URIs, as that is what you will primarily be doing for the rest of the book.

Most of the directives that we will discuss in the remainder of the book, take regular expressions as one of their arguments. And, much of the time, those regular expressions will describe a URI, which is the technical term for the resource that was requested from your server. And most of the time, that means everything after the `http://www.domain.com` part of the web address.

I'll give several common examples of things that you might want to match.

2.14.4 Matching the homepage

Very frequently, people will want to match the home page of the website. Typically, that means that the requested URI is either nothing at all, or is `/`, or is some index page such as `/index.html` or `/index.php`. The case where it is nothing at all would be when the requested address was `http://www.example.com` with no trailing slash.

First, I'll consider the case where they request either `http://www.example.com` or `http://www.example.com/` (ie, with or without the trailing slash, but with no file requested). In other words, we want to match an optional slash.

As you probably remember from earlier, you use the `?` character to make a match optional. Thus, we have: `^/?$`

This matches a string that starts with, and ends with, an optional slash. Or, stated differently, it matches either something that starts and ends with a slash, or something that starts and ends with nothing.

Next, we introduce the additional complexity of the file name. That is, we want to match any of the following four strings:

- The empty string - that is, they requested `http://www.example.com` with no trailing slash.
- `/` - they requested `http://www.example.com/` with a trailing slash.
- `/index.html`
- `/index.php`

We'll build on the regex that we had last time, adding these additional requirements:

```
^/?(index\.(html|php))?$
```

This isn't quite right, as you'll see in a moment, but it's mostly right. It does, however, introduce a new syntax that hasn't been mentioned heretofore. That is the `|` syntax, which has the fancy name of "alternation" and means "one or the other." So `(html|php)` means "either 'html' or 'php'."

So, we've got a regex that means a string that starts with a slash (optional) followed by `index.`, followed by either `html` or `php`, and that entire string (starting with the `index`) is also optional, and then the string ends.

The one problem with this regex is that it also matches the strings `'index.php'` and `'index.html'`, without a leading slash. While, strictly speaking, this is incorrect, in the actual context of matching a URI, it is probably fine, in most scenarios, to ignore that particular technicality. Note, however, that there are lots of people who spend a lot of time trying to figure out how to exploit technicalities like this, so be careful.

2.14.5 Matching a directory

If you wanted to find out what directory a particular requested URI was in, or, perhaps, what keyword it started with, you need to match everything up to the first slash. This will look something like the following:

```
^/([^\/]*)
```

This regex has a number of components. First, there's the standard `^/` which we'll see a lot, meaning "starts with a slash." Following that, we have the character class `[^\/]`, which will match any "not slash" character. This is followed by a `+` indicating that we want one or more of them, and enclosed in parentheses so that we can have the value for later observation, in `$1`.

2.14.6 Matching a filetype

For the third example, we'll try to match everything that has a particular file extension. This, too, is a very common need. For example, we want to match everything that is an image file. The following regex will do that, for the most common image types:

```
\.(jpg|gif|png)$
```

Later on, you'll see how to make this case insensitive, so that files with upper-case file extensions are also matched.

2.15 Regex tools

If you're going to spend more than just a little time messing with regexes, you're eventually going to want a tool that helps you visualize what's going on. There are a number of them available, each of which has different strengths and weaknesses.

2.15.1 Online tools

Online regex testers have become the most popular way to develop and debug regular expressions. They require no installation, and most offer real-time matching, syntax highlighting, and explanations of what your pattern is doing.

regex101

<<https://regex101.com/>> is arguably the best regex testing tool available today. It supports PCRE2, JavaScript, Python, Golang, Java, and .NET regex flavors. It provides a detailed explanation of each part of your pattern, highlights matches and capture groups in real time, and includes a quick reference sidebar. You can save and share your expressions via URL. regex101 is free to use; the source code is not open source, but it runs entirely in the browser after the initial page load and can be used offline.

For the purposes of this book, select the **PCRE2** flavor, as that is the regex engine used by Apache httpd.

RegExr

<<https://regexr.com/>> is another excellent online regex tester. It provides real-time matching, an expression library, and hover-over explanations of pattern components. RegExr is open source ([MIT license](#)).

Regex Pal

<<https://www.regexpal.com/>> is a simpler, no-frills online regex tester. It does one thing — show you what matches — and does it well. Useful when you just want a quick check without the overhead of a full-featured tool.

2.15.2 Command-line tools

pcre2test

pcre2test is a command-line regex tester that ships with the PCRE2 library. On most Linux distributions, you can install it via the `pcre2-utils` or `pcre2-tools` package. On macOS, it is available via Homebrew (`brew install pcre2`).

In addition to telling you whether a particular string matched, it shows what each backreference captures:

```
$ pcre2test
re> /\^\/(\\w+)\(.*\)$/
data> /products/widget-42
 0: /products/widget-42
 1: products
 2: widget-42
```

This is the same PCRE2 engine that Apache httpd uses, so what `pcre2test` tells you is exactly what `mod_rewrite` will do.

Note

If you encounter references to the older `pcrtest` command, that was the PCRE1 version. PCRE1 is end-of-life; `pcr2test` is its replacement.

grep

`grep` is an open source command-line tool (written in Rust, MIT license) that works in the opposite direction — you give it example strings, and it generates a regular expression that matches them. This can be a useful starting point when you know what you want to match but aren't sure how to express it.

2.15.3 Learning resources

If you prefer an interactive tutorial over reading documentation, [RegexLearn](#) provides a step-by-step course that teaches regex by doing. It is open source (MIT license).

2.16 RewriteRule generators

You may find various websites that purport to be RewriteRule generators. I strongly encourage you to avoid these, and instead to learn how to craft your own rules. I've evaluated several of these sites, and every one has resulted in RewriteRule directives that were either enormously inefficient, or completely wrong.

2.17 Summary

Having a good grasp of Regular Expressions is a necessary prerequisite to working with `mod_rewrite`. All too often, people try to build regular expressions by the brute-force method, trying various different combinations at random until something seems to mostly work. This results in expressions that are inefficient and fragile, as well as a great waste of time, and much frustration.

Keep a bookmark in this chapter, and refer back to it when you're trying to figure out what a particular regex is doing.

Other recommended reference sources include the Perl regular expression documentation, which you can find online at <https://perldoc.perl.org/perlre> or by typing `perldoc perlre` at your command line, and the PCRE documentation, which you can find online at <https://www.pcre.org/current/doc/html/>. This is useful even if you're using regex in other implementations (like `mod_rewrite`, for example), since the syntax is largely the same across implementations.

URL MAPPING

Some people are heroes. And some people jot down notes.
Sometimes they're the same person.

—Terry Pratchett, *The Truth*

In this chapter, we'll discuss the various ways that the Apache HTTP Server (httpd) handles URL Mapping. When the Apache HTTP Server receives a request, it is processed in a variety of ways to see what resource it represents. This process is called URL Mapping.

`mod_rewrite` is part of this process, but will be handled separately, since it is a large portion of the contents of this book.

The exact order in which these steps are applied may vary from one configuration to another, so it is important to understand not only the steps, but the way in which you have configured your particular server.

3.1 `mod_rewrite`

`mod_rewrite` is perhaps the most powerful part of this process. That is, of course, why it features prominently in the name of this book.

For now, we'll just say that `mod_rewrite` fills a variety of different roles in the URL mapping process. It can, among other things, modify a URL once it is received, in many different ways.

While this usually happens before the other parts of URL mapping, in certain circumstances, it can also perform that rewriting later on in the process.

This, and much more, will be revealed in the coming chapters.

3.2 DocumentRoot

The `DocumentRoot` directive specifies the filesystem directory from which static content will be served. It's helpful to think of this as the default behavior of the Apache HTTP Server when no other content source is found.

Consider a configuration of the following:

```
DocumentRoot /var/www/html
```

With that setting in place, a request for `<http://example.com/one/two/three.html>` will result in the file `/var/www/html/one/two/three.html` being served to the client with a MIME type derived from the file name - in this case, `text/html`.

3.3 DirectoryIndex

The `DirectoryIndex` directive specifies what file, or files, will be served in the event that a directory is requested. For example, if you have the configuration:

```
DocumentRoot /var/www/html
DirectoryIndex index.html index.php
```

Then when the URL `<http://example.com/one/two/>` is requested, Apache httpd will attempt to serve the file `/var/www/html/index.html` and, if it's not able to find that, will attempt to serve the file `/var/www/html/index.php`.

If neither of those files is available, the next thing it will try to do is serve a directory index.

3.4 FallbackResource

The `FallbackResource` directive, provided by `mod_dir`, defines a default resource to serve when a request doesn't map to any existing file in the filesystem. This is the mechanism behind the “front controller” pattern used by virtually every modern web framework — Laravel, Symfony, WordPress, Drupal, and many others.

Before `FallbackResource` existed (it was introduced in httpd 2.2.16), the standard way to implement a front controller was a `mod_rewrite` rule like this:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ /index.php [L]
```

That four-line incantation — “if it's not an existing file or directory, send it to `index.php`” — appears in countless `.htaccess` files across the web. `FallbackResource` replaces it with a single line:

```
FallbackResource /index.php
```

Existing files — images, CSS, JavaScript, static HTML — are served normally. Only requests that would otherwise produce a 404 are routed to the specified resource. The original request URL is available to the handler via the `REQUEST_URI` server variable.

If the front controller lives in a subdirectory, specify the full path:

```
<Directory "/var/www/html/app">
  FallbackResource /app/index.php
</Directory>
```

You can disable `FallbackResource` in a child directory to prevent inheritance — useful for directories that should return a genuine 404 when a file isn't found:

```
<Directory "/var/www/html/app/static">
  FallbackResource disabled
</Directory>
```

If you find yourself writing a RewriteCond `!-f` / RewriteCond `!-d` pair, stop and consider FallbackResource first. It's simpler, faster (no regex engine involved), and communicates the intent much more clearly. Save mod_rewrite for cases where you need to transform the URL, not merely route it.

3.5 Automatic directory listings

The module mod_autoindex serves a file listing for any directory that doesn't contain a DirectoryIndex file. (See section *DirectoryIndex* above.)

To permit directory listings, you must enable the Indexes setting of the Options directive:

```
Options +Indexes
```

See the documentation of the Options <<https://httpd.apache.org/docs/current/mod/core.html#options>> for further discussion of that directive.

If the Indexes option is on, then a directory listing will be displayed, with whatever features are enabled by the IndexOptions directive.

Typically, a directory will look like the example shown below.

Index of /files

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 Contents/	2013-04-25 21:58	-	
 DSCN1259.JPG	2013-04-25 21:58	431K	
 Door.JPG	2013-04-25 21:58	2.7M	
 NewIn2.4.key	2013-04-25 21:57	3.4M	
 images/	2013-04-25 21:57	-	
 index.xml.gz	2013-04-25 21:58	38K	
 thumbs/	2013-04-25 21:58	-	

For further discussion of the autoindex functionality, consult the `mod_autoindex` documentation at https://httpd.apache.org/docs/current/mod/mod_autoindex.html.

Future versions of this book will include more detailed information about directory listings.

3.6 Alias

The `Alias` directive is used to map a URL to a directory path outside of your `DocumentRoot` directory.

```
Alias /icons /var/www/icons
```

An `Alias` is usually accompanied by a `<Directory>` stanza granting `httpd` permission to look in that directory. In the case of the above `Alias`, for example, add the following:

```
<Directory /var/www/icons>  
  Require all granted  
</Directory>
```

Or, if you're using `httpd 2.2` or earlier:

```
<Directory /var/www/icons>  
  Order allow,deny  
  Allow from all  
</Directory>
```

There's a special form of the `Alias` directive - `ScriptAlias` - which has the additional property that any file found in the referenced directory will be assumed to be a CGI program, and `httpd` will attempt to execute it and sent the output to the client.

CGI programming is outside of the scope of this book. You may read more about it at <https://httpd.apache.org/docs/current/howto/cgi.html>

3.7 AliasMatch and ScriptAliasMatch

`AliasMatch` and `ScriptAliasMatch` are regex-capable versions of `Alias` and `ScriptAlias`. They allow you to use regular expressions to match the URL and use backreferences in the target path.

For example, to map user CGI directories without `mod_userdir`:

```
ScriptAliasMatch "^/~([a-zA-Z0-9]+)/cgi-bin/(.+)" "/home/$1/cgi-bin/$2"
```

A request for `http://example.com/~alice/cgi-bin/stats.pl` would execute `/home/alice/cgi-bin/stats.pl`.

Similarly, `AliasMatch` can map patterns of URLs to filesystem locations:

```
AliasMatch "^/docs/([a-z]{2})/" "/srv/docs/$1/"
```

This maps `/docs/en/` to `/srv/docs/en/`, `/docs/fr/` to `/srv/docs/fr/`, and so on.

A common gotcha: unlike `Alias`, which treats the URL prefix as a literal string, `AliasMatch` consumes the entire URL-path during the match. If you're not careful with your regex, you may capture more or less than you intended. Use anchors (^ and \$) and be deliberate about what your groups capture.

3.8 Redirect

The purpose of the `Redirect` directive is to cause a requested URL to result in a redirection to a different resource, either on the same website or on a different server entirely.

The `Redirect` directive results in a `Location` header, and a 30x status code, being sent to the client, which will then make a new request for the specified resource.

The exact value of the 30x status code will influence what the client does with this information, as indicated in the table below:

Code	Meaning
300	Multiple Choice - Several options are available
301	Moved Permanently
302	Temporary Redirect
304	Not Modified - use whatever version you have cached

Other 30x statuses are available, but these are the only ones we'll concern ourselves with at the moment.

The syntax of the `Redirect` directive is as follows:

```
Redirect [status] RequestedURL TargetUrl
```

3.9 RedirectMatch

`RedirectMatch` is the regex-capable counterpart to `Redirect`. It allows you to match the requested URL against a regular expression and use backreferences in the target URL.

For example, to redirect an entire directory tree while preserving the path structure:

```
RedirectMatch 301 "^/oldsite/(.*)" "https://newsite.example.com/$1"
```

This redirects `/oldsite/page.html` to `https://newsite.example.com/page.html`, and so on for any path under `/oldsite/`.

Another common use is stripping or adding file extensions:

```
RedirectMatch 301 "^/(.+)\.htm$" "/$1.html"
```

`RedirectMatch` is often a simpler and more appropriate choice than a `RewriteRule` with the `[R]` flag when all you need is a pattern-based redirect. It doesn't require `RewriteEngine On` and it expresses the intent — “redirect” — directly.

3.10 Location

The `<Location>` directive limits the scope of the enclosed directives by URL. It is similar to the `<Directory>` directive, and starts a subsection which is terminated with a `</Location>` directive. `<Location>` sections are processed in the order they appear in the configuration file, after the `<Directory>` sections and `.htaccess` files are read, and after the `<Files>` sections.

`<Location>` sections operate completely outside the filesystem. This has several consequences. Most importantly, `<Location>` directives should not be used to control access to filesystem locations. Since several different URLs may map to the same filesystem location, such access controls may be circumvented.

The enclosed directives will be applied to the request if the path component of the URL meets any of the following criteria:

The specified location matches exactly the path component of the URL. The specified location, which ends in a forward slash, is a prefix of the path component of the URL (treated as a context root). The specified location, with the addition of a trailing slash, is a prefix of the path component of the URL (also treated as a context root). In the example below, where no trailing slash is used, requests to `/private1`, `/private1/` and `/private1/file.txt` will have the enclosed directives applied, but `/private1other` would not.

```
<Location /private1>
# ...
</Location>
```

In the example below, where a trailing slash is used, requests to `/private2/` and `/private2/file.txt` will have the enclosed directives applied, but `/private2` and `/private2other` would not.

```
<Location /private2/>
# ...
</Location>
```

When to use `<Location>` Use `<Location>` to apply directives to content that lives outside the filesystem. For content that lives in the filesystem, use `<Directory>` and `<Files>`. An exception is `<Location />`, which is an easy way to apply a configuration to the entire server. For all origin (non-proxy) requests, the URL to be matched is a URL-path of the form `/path/`. No scheme, hostname, port, or query string may be included. For proxy requests, the URL to be matched is of the form `scheme://servername/path`, and you must include the prefix.

The URL may use wildcards. In a wild-card string, `?` matches any single character, and `*` matches any sequences of characters. Neither wildcard character matches a `/` in the URL-path.

Regular expressions can also be used, with the addition of the `~` character. For example:

```
<Location ~ "/(extra|special)/data">
#...
</Location>
```

would match URLs that contained the substring `/extra/data` or `/special/data`. The directive `<LocationMatch>` behaves identically to the regex version of `<Location>`, and is preferred, for the simple reason that `~` is hard to distinguish from `-` in many fonts, leading to configuration errors when you're following examples.

```
<LocationMatch "/(extra|special)/data">
  #...
</LocationMatch>
```

The <Location> functionality is especially useful when combined with the SetHandler directive. For example, to enable status requests, but allow them only from browsers at example.com, you might use:

```
<Location /status>
  SetHandler server-status
  Require host example.com
</Location>
```

3.10.1 Virtual Hosts

Rather than running a separate physical server, or separate instance of httpd, for each website, it is common practice to run sites via virtual hosts. Virtual hosting refers to running more than one web site on the same web server.

Virtual hosts can be name-based - that is, multiple hostnames resolving to the same IP address - or IP based - that is, a dedicated IP address for each site - depending on various factors including availability of IP addresses and preference. Name-based virtual hosting is more common, but there are scenarios in which IP-based hosting may be preferred.

Virtual hosting is discussed in more detail in *Virtual hosts and mod_rewrite*.

3.11 mod_vhost_alias

When you have a handful of virtual hosts, writing an explicit <VirtualHost> block for each one is straightforward. When you have hundreds or thousands — as a hosting provider might — individual blocks become unmanageable. mod_vhost_alias solves this by dynamically deriving the document root from the hostname of the incoming request.

The simplest configuration uses VirtualDocumentRoot with the %0 interpolation token, which expands to the full server name:

```
UseCanonicalName Off
VirtualDocumentRoot "/var/www/vhosts/%0"
```

A request for `http://www.example.com/page.html` is served from `/var/www/vhosts/www.example.com/page.html`. No per-host configuration is needed — just create the directory and drop in the files.

More sophisticated interpolation tokens let you split the hostname into components. %1 is the first dot-separated part, %2 the second, %-1 the last, and so on. You can even extract individual characters for hash-based directory layouts:

```
VirtualDocumentRoot "/var/www/vhosts/%3+/%2.1/%2.2/%2.3/%2"
```

This maps `www.domain.example.com` to `/var/www/vhosts/example.com/d/o/m/domain/`.

There's also `VirtualScriptAlias` and `VirtualScriptAliasIP` for CGI directories, and `VirtualDocumentRootIP` for IP-based mass hosting.

Before reaching for `mod_rewrite` to implement mass virtual hosting, check whether `mod_vhost_alias` does what you need — it's faster and far simpler to maintain.

3.11.1 Proxying

`mod_proxy` and its family of protocol-specific sub-modules (`mod_proxy_http`, `mod_proxy_fcgi`, `mod_proxy_ajp`, `mod_proxy_wstunnel`, and others) allow `httpd` to forward requests to another server and return the response to the client. This is a form of URL mapping — the URL is mapped not to a local file but to a remote resource.

The most common directive is `ProxyPass`, which maps a local URL prefix to a backend:

```
ProxyPass "/app" "http://appserver.local:8080/app"
ProxyPassReverse "/app" "http://appserver.local:8080/app"
```

`ProxyPassReverse` rewrites `Location` headers in the backend's response so that redirects point to the proxy's URL rather than the backend's. Without it, clients may be redirected to URLs they can't reach.

Proxying interacts with `mod_rewrite` via the `[P]` flag, which is discussed in *Proxies and mod_rewrite*. The short version: `[P]` causes a `RewriteRule` substitution to be treated as a proxy request. This is powerful but has subtleties — and in many cases a simple `ProxyPass` is both clearer and more efficient.

3.12 mod_proxy_express

`mod_proxy_express` does for reverse proxying what `mod_vhost_alias` does for document roots: it dynamically maps incoming hostnames to backend URLs using a DBM file, without requiring per-host configuration.

A simple text file maps hostnames to backends:

```
www1.example.com http://192.168.211.2:8080
www2.example.com http://192.168.211.12:8088
www3.example.com http://192.168.212.10
```

Convert it to DBM with `httxt2dbm`, then enable the module:

```
ProxyExpressEnable on
ProxyExpressDBMFile /etc/httpd/proxy-map
```

This is a lightweight alternative to using `RewriteMap` with the `[P]` flag for dynamic reverse proxying.

3.12.1 mod_actions

`mod_actions` lets you trigger a CGI script based on the MIME type of the requested resource or the HTTP request method.

The `Action` directive maps a handler or MIME type to a CGI script:

```
Action image/gif /cgi-bin/image-handler.cgi
```

Any request for a `.gif` file will be handled by `/cgi-bin/image-handler.cgi`, which receives the original URL in the `PATH_INFO` and `PATH_TRANSLATED` environment variables.

You can also fire a script for a specific HTTP method using the `Script` directive:

```
Script PUT /cgi-bin/upload-handler.cgi
```

This is a niche feature, but when you need it, it's simpler than trying to match request methods with `mod_rewrite`.

3.12.2 mod_imagemap

`mod_imagemap` provides server-side image map processing — an early web technology where different regions of an image could link to different URLs. Clicking on a specific area of the image sends the coordinates to the server, which looks them up in a map file and returns the appropriate URL.

While server-side image maps have been almost entirely replaced by client-side image maps (the HTML `<map>` and `<area>` elements) and modern JavaScript-driven interfaces, `mod_imagemap` remains part of the httpd distribution for backwards compatibility.

3.12.3 mod_negotiation

`mod_negotiation` implements content negotiation — the ability for the server to choose the best representation of a resource based on the client's stated preferences (language, media type, encoding, character set).

The most visible feature is `MultiViews`, enabled via the `Options` directive:

```
Options +MultiViews
```

With `MultiViews` enabled, a request for `/docs/manual` causes httpd to search for files matching the pattern `/docs/manual.*` and choose the best match based on the `Accept-*` headers in the request. So if both `manual.en.html` and `manual.fr.html` exist, a French browser will receive the French version.

A more explicit approach uses type maps — files (typically with a `.var` extension) that list the available variants and their properties:

```
URI: manual

URI: manual.en.html
Content-Type: text/html
Content-Language: en

URI: manual.fr.html
Content-Type: text/html
Content-Language: fr
```

Content negotiation is worth understanding because it can interact with `mod_rewrite` in surprising ways. If `MultiViews` is on and you have a rewrite rule that expects a literal file path, the negotiation phase may

match a file before your rule fires — or your rule may fire and then negotiation remaps the result. When debugging unexpected behavior, check whether MultiViews is enabled.

3.12.4 mod_userdir

`mod_userdir` enables the classic Unix convention of per-user web directories accessed via `http://example.com/~username/`. The `UserDir` directive specifies which directory within a user's home directory serves as their web root:

```
UserDir public_html
```

A request for `http://example.com/~alice/index.html` is then served from `/home/alice/public_html/index.html`.

You can also point `UserDir` at an entirely different directory tree:

```
UserDir /var/www/users
```

This maps `~alice` to `/var/www/users/alice/`, regardless of where Alice's home directory actually is.

For security, you should typically disable `UserDir` for sensitive accounts — especially root:

```
UserDir disabled root
```

You can also take the whitelist approach — disable everyone and explicitly enable specific users:

```
UserDir disabled
UserDir enabled alice bob carol
```

The `~` in URLs can be awkward, and some administrators prefer cleaner paths like `/users/alice/`. This is achievable with an `AliasMatch` (see *AliasMatch and ScriptAliasMatch* above) and doesn't require `mod_userdir` at all.

3.12.5 mod_speling

`mod_speling`¹ attempts to fix mistyped URLs by performing a case-insensitive match and allowing up to one character error — an insertion, omission, transposition, or wrong character.

Enable it with:

```
CheckSpelling On
```

Note that the directive is `CheckSpelling` — with two l's.²

If a request for `/Index.HTML` doesn't find a file but `/index.html` exists, `mod_speling` will issue a 301 redirect to the correct URL. If multiple close matches exist, the client receives a 300 (Multiple Choices) response listing the candidates.

Use `CheckCaseOnly On` to limit correction to capitalization differences without attempting to fix other misspellings.

¹ Yes, with one "l" — because it's hilarious, see?

² Just to keep you on your toes.

Caveats:

- `mod_speling` performs a directory scan for each miss, which can be expensive on busy servers or large directories.
- It may match files you didn't intend — for example, correcting a request for `/status` to `/stats.html` when you meant the `server-status` handler.
- It should not be enabled in DAV-enabled directories, where it can redirect write operations to unintended resources.

`mod_speling` can eliminate a surprising number of 404 errors caused by case differences — particularly useful when migrating from a case-insensitive filesystem (Windows/IIS) to a case-sensitive one (Linux). It's a lighter touch than writing `RewriteRule` patterns to handle every possible capitalization variant.

3.12.6 File not found

In the event that a requested resource is not available, after all of the above mentioned methods are attempted to find it, `httpd` returns a 404 (Not Found) response. The `ErrorDocument` directive lets you customize what the client sees:

```
ErrorDocument 404 /errors/not-found.html
```

The argument can be a local URL-path (as above), an external URL, or a simple text string (prefixed with a double-quote character):

```
ErrorDocument 404 "Sorry, we couldn't find that page."
```

`ErrorDocument` works for any HTTP status code, not just 404 — you can customize 403 (Forbidden), 500 (Internal Server Error), and others.

Note that `FallbackResource` (see [FallbackResource](#) above) fires *before* the 404 would be generated. If `FallbackResource` is set, `ErrorDocument 404` will only trigger for requests that your fallback handler itself decides to reject.

INTRODUCTION TO MOD_REWRITE

In the high and far-off times the Elephant, O Best Beloved, had no trunk.

—Rudyard Kipling, *The Elephant's Child*

`mod_rewrite` is the power tool of Apache httpd URL mapping. Of course, sometimes you just need a screwdriver, but when you need the power tool, it's good to know where to find it.

`mod_rewrite` provides sophisticated URL manipulation via regular expressions, and the ability to do a variety of transformations, including, but not limited to, modification of the request URL. You can additionally return a variety of status codes, set cookies and environment variables, proxy requests to another server, or send redirects to the client.

In this chapter we'll cover `mod_rewrite` syntax and usage, and in the next chapter we'll give a variety of examples of using `mod_rewrite` in common scenarios.

4.1 Where `mod_rewrite` fits in the request lifecycle

Before looking at configuration details, it helps to understand *when* `mod_rewrite` runs during request processing. Apache httpd handles requests in a series of phases — URL-to-filename translation, authentication, authorization, fixup, content generation, and logging. `mod_rewrite` hooks into two of these:

1. **URL-to-filename translation** — where server and `<VirtualHost>` context rules run. This happens early, before authentication.
2. **Fixup** — where *per-directory context* rules run (from `.htaccess` files and `<Directory>` blocks). By this point the URL has already been mapped to a filesystem path, which is why per-directory rules see a stripped path rather than the full URL.

The following diagram shows the full pipeline and where `mod_rewrite`'s two hooks sit:

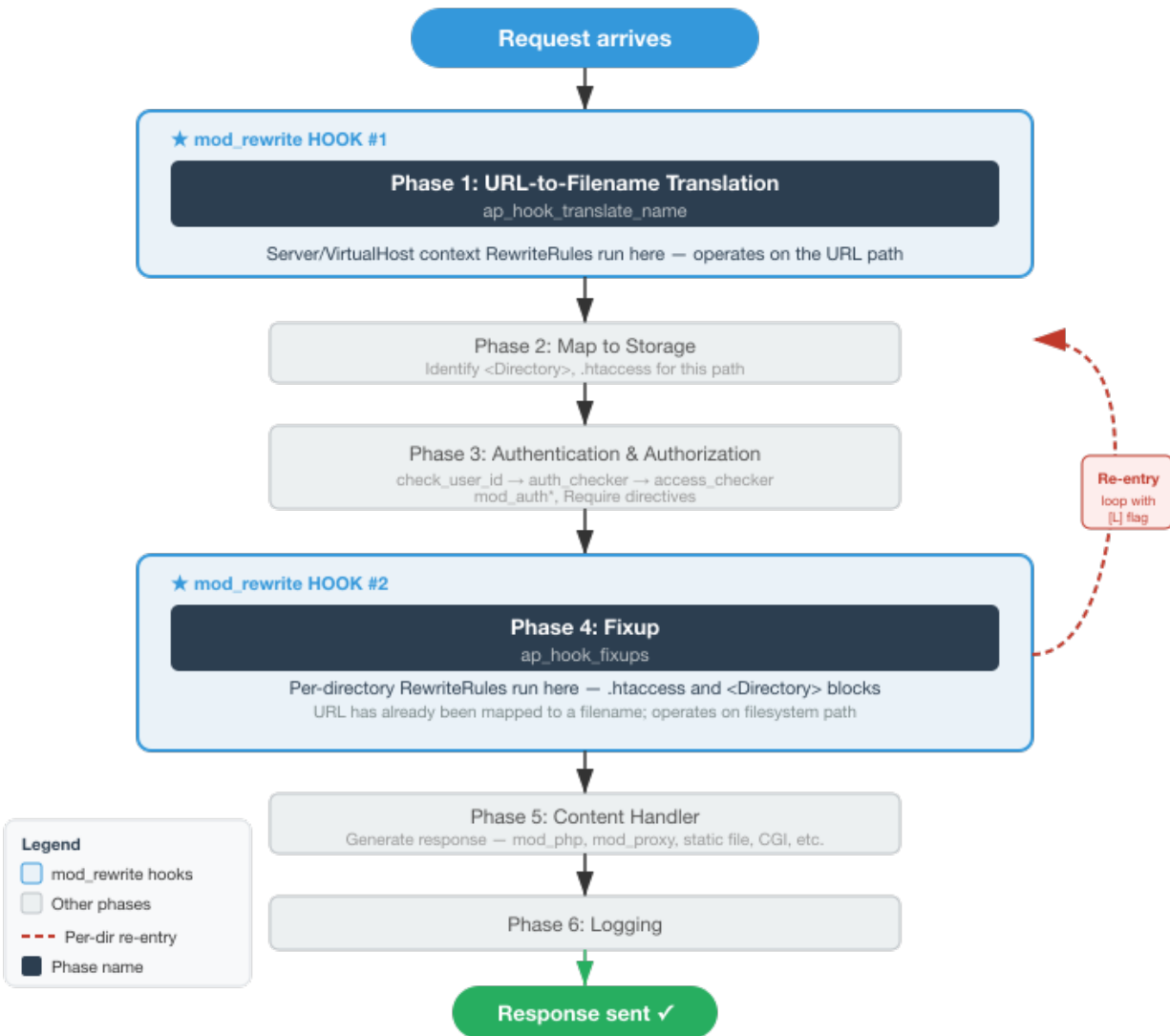
This two-phase design explains many of the differences between server-context and per-directory rules that the rest of this chapter covers — particularly the path-stripping behavior in `.htaccess` files and the re-entry loop that the `[L]` and `[END]` flags control (see *RewriteRule Flags*).

4.1.1 Loading `mod_rewrite`

To use `mod_rewrite` in any context, you need to have the module loaded. If you're the server administrator, this means having the following line somewhere in your Apache httpd configuration:

Apache httpd Request Processing Phases

Where mod_rewrite hooks in: server context vs. per-directory context



```
LoadModule rewrite_module modules/mod_rewrite.so
```

This tells httpd that it needs to load `mod_rewrite` at startup time, so as to make its functionality available to your configuration files.

If you are not the server administrator, then you'll need to ask your server administrator if the module is available, or experiment to see if it is. If you're not sure, you can test to see whether it's enabled in the following manner.

Create a subdirectory in your document directory. Let's call it `test_rewrite`

Create a file in that directory called `.htaccess` and put the following text in it:

```
RewriteEngine on
```

Create another file in that directory called `index.html` containing the following text:

```
<html>
Hello, mod_rewrite
</html>
```

Now, point your browser at that location:

```
http://example.com/test_rewrite/index.html
```

You'll see one of two things. Either you'll see the words `Hello, mod_rewrite` in your browser, or you'll see the ominous words `Internal Server Error`. In the former case, everything is fine - `mod_rewrite` is loaded and your `.htaccess` file worked just fine. If you got an `Internal Server Error`, that was httpd complaining that it didn't know what to do with the `RewriteEngine` directive, because `mod_rewrite` wasn't loaded.

If you have access to the server's error log file, you'll see the following in it:

```
Invalid command 'RewriteEngine', perhaps misspelled or defined by a module not
↪included in the server configuration
```

Which is httpd's way of saying that you used a directive (`RewriteEngine`) without first loading the module that defines that directive.

If you see the `Internal Server Error` message, or that log file message, it's time to contact your server administrator and ask if they'll load `mod_rewrite` for you.

However, this is fairly unlikely, since `mod_rewrite` is a fairly standard part of any Apache HTTP Server's bag of tricks.

4.1.2 RewriteEngine

In the section above, we used the `RewriteEngine` directive without defining what it does.

The `RewriteEngine` directive enables or disables the runtime rewriting engine. The directive defaults to `off`, so the result is that rewrite directives will be ignored in any scope where you don't have the following:

RewriteEngine On

While we won't always include that in every example in this book, it should be assumed, from this point forward, that every use of `mod_rewrite` occurs in a scope where `RewriteEngine` has been turned on.

4.2 How `mod_rewrite` interacts with other modules

A common source of confusion is mixing `mod_rewrite` rules with `mod_alias` directives (`Alias`, `Redirect`, `ScriptAlias`, `RedirectMatch`) or `mod_proxy` directives (`ProxyPass`) in the same configuration scope. The results can be surprising if you don't know the processing order.

In the URL-to-filename translation phase:

1. **`mod_rewrite` runs first.** Your `RewriteRule` directives are evaluated before any `Alias` or `Redirect` directives.
2. **`mod_alias` runs second.** `Redirect` and `RedirectMatch` directives can still fire if the rewritten URL matches them — and in some contexts, `Redirect` can preempt a `RewriteRule` via early return.
3. **`mod_proxy` runs in a separate phase.** `ProxyPass` is not part of URL-to-filename translation. Mixing `RewriteRule [P]` with `ProxyPass` for the same path can cause conflicts.

If you need a `RewriteRule` substitution to be passed through to `mod_alias` for further processing (rather than being treated as a final filesystem path), use the `[PT]` (`passthrough`) flag. Without it, rewritten URLs bypass `mod_alias` entirely.

The following diagram illustrates the interaction:

4.2.1 `mod_rewrite` in `.htaccess` files

Before we go any further, it's critical to note that things are different, in several important ways, if you have to use `.htaccess` files for configuration.

4.3 What are `.htaccess` files?

`.htaccess` files are per-directory configuration files, for use by people

who don't have access to the main server configuration file. For the most part, you put configuration directives into `.htaccess` files just as you would in a `<Directory>` block in the server configuration, but there are some differences.

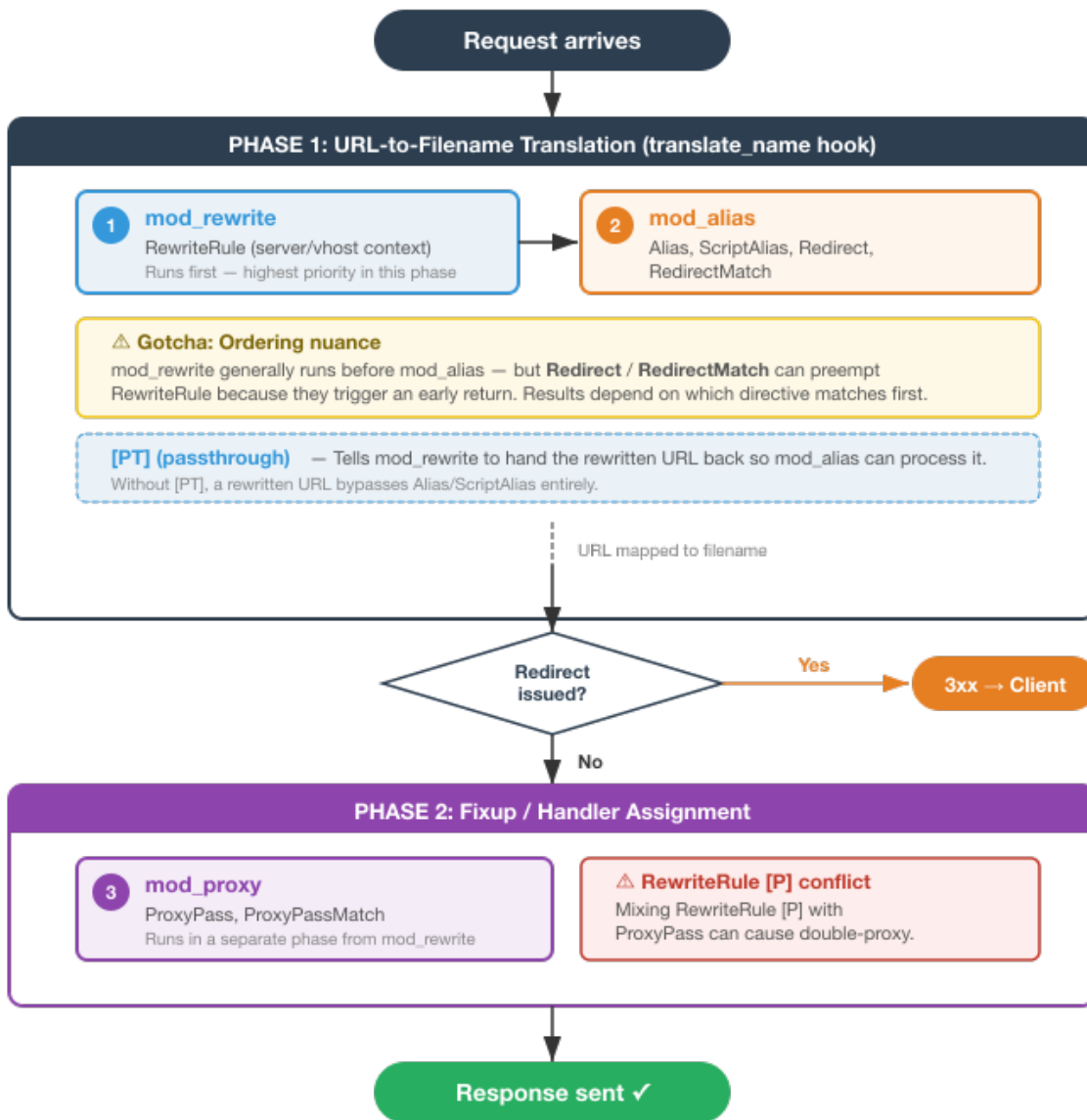
The most important of these differences is that the `.htaccess` file is consulted every time a resource is requested from the directory in question, whereas configurations placed in the main server configuration file are loaded once, at server startup.

The positive side of this is that you can modify the contents of a `.htaccess` file and have the change take effect immediately, as of the next request received by the server.

The negative is that the `.htaccess` file needs to be loaded from the filesystem on every request, resulting in an incremental slowdown for every request. Additionally, because `httpd` doesn't know ahead of time what directories contain `.htaccess` files, it has to look in each directory for them, all along the path to the requested resource, which results in a slowdown that grows with the depth of the directory tree.

Module Processing Order: mod_rewrite + mod_alias + mod_proxy

How three modules interact in the same configuration scope



Key takeaways:

- **mod_rewrite** runs first in translate_name — a RewriteRule can preempt Alias/Redirect.
- **Redirect / RedirectMatch** can still "win" if they match before rewriting completes.
- Use **[PT]** to pass a rewritten URL through to mod_alias (Alias, ScriptAlias).
- **ProxyPass** runs in a different phase — don't mix it with RewriteRule [P] for the same path.
- Debug with `LogLevel rewrite:trace3` and `httpd -s` to verify processing order.

"If you're mixing Redirect/RedirectMatch with RewriteRule in the same context, the processing order may not be what you expect."

In Apache httpd 2.2 and earlier, `.htaccess` files are enabled by default - that is the configuration directive that enables them, `AllowOverride`, has a default value of `All`. In 2.4 and later, it has a default value of `None`, so `.htaccess` files are disabled by default.

A typical configuration to permit the use of `.htaccess` files looks like:

```
<Directory />
  AllowOverride None
</Directory>
DocumentRoot /var/www/html
<Directory /var/www/html>
  AllowOverride All
  Options +FollowSymLinks
</Directory>
```

That is to say, `.htaccess` files are disallowed for the entire filesystem, starting at the root, but then are permitted in the document directories. This prevents httpd from looking for `.htaccess` files in `/`, `/var`, and `/var/www` on the way to looking in `/var/www/html`.¹

Note that in order to enable the use of `mod_rewrite` directives in `.htaccess` files, you also need to enable `Options FollowSymLinks`. A `RewriteRule` may be thought of as a kind of symlink, because it allows you to serve content from other directories via a rewrite. Thus, for reasons of security, it is necessary to enable symlinks in order to use `mod_rewrite`.

4.4 Ok, so, what's the deal with mod_rewrite in .htaccess files?

There are two major differences that you must be aware of before we proceed any further. The exact implications of these differences will become more apparent as we go, but I wouldn't want them to surprise you.

First, there are two directives that you cannot use in `.htaccess` files. These directives are `RewriteMap` and (prior to httpd 2.4) `RewriteLog`. These must be defined in the main server configuration. The reasons for this will be discussed in greater length when we get to the sections about those directives `RewriteMap` and `RewriteLogging`, respectively).

Second, and more importantly, the syntax of `RewriteRule` directives changes in `.htaccess` context in a way that you'll need to be aware of every time you write a `RewriteRule`. Specifically, the directory path that you're in will be removed from the URL path before it is presented to the `RewriteRule`.

The exact implications of this will become clearer as we show you examples. And, indeed, every example in this book will be presented in a form for the main config, and a form for `.htaccess` files, whenever there is a difference between the two forms. But we'll start with a simple example to illustrate the idea.

Some of this, you'll need to take on faith at the moment, since we've not yet introduced several of the concepts presented in this example, so please be patient for now.

Consider a situation where you want to apply a rewrite to content in the `/images/puppies/` subdirectory of your website. You have four options: You can put the `RewriteRule` in the main server configuration file; You can place it in a `.htaccess` file in the root of your website; You can place it in a `.htaccess` file in the `images` directory; Or you can place it in a `.htaccess` file in the `images/puppies` directory.

¹ Or, more to the point, it prevents malicious end-users from finding ways to look there.

Here's what the rule might look like in those various scenarios:

Location	Rule
Main config	<code>RewriteRule ^/images/puppies/(.*)\.jpg /dogs/\$1.gif</code>
Root directory	<code>RewriteRule ^images/puppies/(.*)\.jpg /dogs/\$1.gif</code>
images directory	<code>RewriteRule ^puppies/(.*)\.jpg /dogs/\$1.gif</code>
images/puppies directory	<code>RewriteRule ^(.*)\.jpg /dogs/\$1.gif</code>

For the moment, don't worry too much about what the individual rules do. Look instead at the URL path that is being considered in each rule, and notice that for each directory that a `.htaccess` file is placed in, the directory path that `RewriteRule` may consider is relative to that directory, and anything above that becomes invisible for the purpose of `mod_rewrite`.

Don't worry too much if this isn't crystal clear at this point. It will become more clear as we proceed and you see more examples.

4.5 So, what do I do?

If you don't have access to the main server configuration file, as it the case for many of the readers of this book, don't despair. `mod_rewrite` is still a very powerful tool, and can be persuaded to do almost anything that you need it to do. You just need to be aware of its limitations, and adjust accordingly when presented with an example rule.

We aim to help you do that at each step along this journey.

4.5.1 RewriteOptions

The `RewriteOptions` directive controls several special behaviors of the rewrite engine. You can specify multiple options separated by spaces.

4.6 Inherit and InheritBefore

By default, rewrite rules are *not* inherited from parent contexts. A `<VirtualHost>` does not inherit rules from the main server config; a `.htaccess` file does not inherit rules from a parent directory's `.htaccess`.

`RewriteOptions Inherit` forces the current context to inherit the parent's rules, maps, and conditions. The inherited rules run *after* the local rules.

`RewriteOptions InheritBefore` does the same, but the inherited rules run *before* the local rules.

```
# In a .htaccess or <Directory> block:
```

```
RewriteOptions Inherit
```

4.7 InheritDown and InheritDownBefore

These are the inverse of `Inherit`: instead of a child saying “give me my parent’s rules,” the parent says “push my rules into all children.” This avoids needing `RewriteOptions Inherit` in every child configuration.

`InheritDown` pushes the parent’s rules to run *after* each child’s local rules. `InheritDownBefore` pushes them to run *before*.

2.4.8 Available in httpd 2.4.8 and later.

4.8 IgnoreInherit

If a parent has `InheritDown` set but a particular child should *not* inherit, the child can use `RewriteOptions IgnoreInherit` to opt out.

2.4.8 Available in httpd 2.4.8 and later.

4.9 AllowNoSlash

By default, `mod_rewrite` ignores URLs that map to a directory on disk but lack a trailing slash — it assumes `mod_dir` will handle the redirect. If you’ve set `DirectorySlash Off`, enable `AllowNoSlash` so that rewrite rules can match directory URLs without a trailing slash.

2.4 Available in httpd 2.4.0 and later.

4.10 AllowAnyURI

In `server/vhost` context (since httpd 2.2.22), `mod_rewrite` only processes requests whose URI is a valid URL-path. This is a security measure (see CVE-2011-3368 and CVE-2011-4317). `AllowAnyURI` lifts that restriction.

Warning

Enabling this makes the server vulnerable to security issues if rewrite rules are not carefully authored. Use with extreme caution.

2.4.3 Available in httpd 2.4.3 and later.

4.11 MergeBase

Copies the value of `RewriteBase` from where it’s explicitly defined into any sub-directory or sub-location that doesn’t define its own. This was the default behavior in httpd 2.4.0–2.4.3; the option restores it.

2.4.4 Available in httpd 2.4.4 and later.

4.12 IgnoreContextInfo

When a relative substitution is made in per-directory context and RewriteBase has not been set, mod_rewrite uses extended URL and filesystem context information (provided by modules like mod_userdir and mod_alias) to resolve the substitution back into a URL. This option disables that behavior.

2.4.16 Available in httpd 2.4.16 and later.

4.13 LegacyPrefixDocRoot

Prior to 2.4.26, when a substitution was an absolute URL matching the current virtual host, the URL could be reduced to a local path and the document root would be prepended. This option restores that legacy behavior.

2.4.26 Available in httpd 2.4.26 and later.

4.14 LongURLOptimization

Reduces memory usage for long, unoptimized rule sets that repeatedly expand long values in RewriteCond and RewriteRule variables.

trunk Available in httpd trunk (future 2.5.x) only — not yet in any stable release.

4.14.1 RewriteBase

The RewriteBase directive sets the base URL for per-directory rewrites. It is only valid in per-directory context (.htaccess files and <Directory> blocks) and is ignored in server or virtual host context.

When mod_rewrite processes a rule in .htaccess, it strips the local directory prefix from the URL before matching, then prepends it back after substitution. RewriteBase overrides what gets prepended.

Consider a .htaccess file in /var/www/html/app/, where the URL /app/ maps to that directory:

```
# /var/www/html/app/.htaccess
RewriteEngine On
RewriteBase /app/
RewriteRule ^page/(.*)$ index.php?page=$1 [L]
```

Without RewriteBase /app/, the substitution index.php?page=foo would be interpreted relative to the filesystem path, not the URL path, and the result might not be what you expect.

The most common value is simply:

```
RewriteBase /
```

This tells mod_rewrite that all substitutions should be treated as relative to the document root.

When do you need RewriteBase?

- In .htaccess files when your rewrite substitutions are relative paths (not starting with /).

- When the URL path to the directory containing the `.htaccess` differs from the filesystem path (e.g., due to `Alias`).
- You do *not* need it in `<VirtualHost>` or server config — there, `RewriteRule` operates on the full URL-path and no prefix stripping occurs.

When can you omit it?

- When all your substitutions use absolute URL-paths (starting with `/`).
- When the URL-to-filesystem mapping is straightforward (`DocumentRoot + URL-path = filesystem path`).

A common source of confusion: people put `RewriteBase` in server config or `<VirtualHost>` blocks where it has no effect, then wonder why their rules behave unexpectedly. If you're not in a `.htaccess` or `<Directory>` context, you don't need it.

REWRITERULE

Quickly, bring me a beaker of wine, that I may wet
my brain and say something clever.

—Aristophanes (attributed)

I'll start the main technical discussion of `mod_rewrite` with the `RewriteRule` directive, as it is the workhorse of `mod_rewrite`, and the directive that you'll encounter most frequently.

`RewriteRule` performs manipulation of a requested URL, and along the way can do a number of additional things. It's where the actual rewriting happens — everything else in `mod_rewrite` exists to support it.

The syntax of a `RewriteRule` is fairly simple, but you'll find that exploring all of the possible permutations of it will take a while. So I'll provide a lot of examples along the way to illustrate.

If you learn best by example, you may want to jump back and forth between this section and *Recipes* to help you make sense of this all.

5.1 Syntax

A `RewriteRule` directive has two required arguments and optional flags. It looks like:

```
RewriteRule PATTERN TARGET [FLAGS]
```

The following sections will discuss each of those arguments in great detail, but these are defined as:

PATTERN

A regular expression to be applied to the requested URI.

TARGET

What the URI will be rewritten to.

FLAGS

Optional flags that modify the behavior of the rule.

5.2 Pattern

The `PATTERN` argument of the `RewriteRule` is a regular expression that is applied to the URL path, or file path, depending on the context.

In VirtualHost context, or in server-wide context, PATTERN will be matched against the part of the URL after the hostname and port, and before the query string (the %-decoded URL-path). For example, in the URL `<http://example.com/dogs/index.html?dog=collie>`, the pattern will be matched against `/dogs/index.html`.

In *per-directory context* — that is, within a `<Directory>`, `<DirectoryMatch>`, `<Files>`, or `<FilesMatch>` section, or in a `.htaccess` file — PATTERN will be matched against the filesystem path, after removing the prefix that led the server to the current RewriteRule (e.g. either “`dogs/index.html`” or “`index.html`” depending on where the directives are defined). See *Per-directory context gotchas* below for the gory details of how this prefix stripping works — it’s one of the most common sources of confusion.

Subsequent RewriteRule patterns are matched against the output of the last matching RewriteRule.

It is assumed, at this point, that you’ve already read the chapter Introduction to Regular Expressions, and/or are familiar with what a regular expression is, and how to craft one.

5.2.1 Negated patterns

You can prefix the pattern with an exclamation mark (!) to negate it. This means the rule fires when the URL does *not* match the pattern. I find this useful for “everything except” rules — for example, redirecting all requests that are *not* for a specific path:

```
# Redirect everything that ISN'T the maintenance page
RewriteRule !^maintenance\.html$ /maintenance.html [R=302,L]
```

There’s one important caveat: when you negate a pattern, there’s nothing to capture. The pattern didn’t match, so there are no groups, and \$1, \$2, etc. are empty. If you need backreferences in the target *and* you need a negated match, use a RewriteCond instead:

```
# This does NOT work - $1 is empty because the pattern is negated
RewriteRule !^secret/ /public/$1 [L]

# Do this instead
RewriteCond %{REQUEST_URI} !^/secret/
RewriteRule ^(.*)$ /public/$1 [L]
```

See *RewriteCond* for more on conditions.

5.3 Target

The target of a RewriteRule can be one of the following:

5.3.1 A file-system path

Designates the location on the file-system of the resource to be delivered to the client. Substitutions are only treated as a file-system path when the rule is configured in server (virtualhost) context and the first component of the path in the substitution exists in the file-system

5.3.2 URL-path

A DocumentRoot-relative path to the resource to be served. Note that `mod_rewrite` tries to guess whether you have specified a file-system path or a URL-path by checking to see if the first segment of the path exists at the root of the file-system. For example, if you specify a Substitution string of `/www/file.html`, then this will be treated as a URL-path unless a directory named `www` exists at the root or your file-system (or, in the case of using rewrites in a `.htaccess` file, relative to your document root), in which case it will be treated as a file-system path. If you wish other URL-mapping directives (such as `Alias`) to be applied to the resulting URL-path, use the `[PT]` flag as described below.

5.3.3 Absolute URL

If an absolute URL is specified, `mod_rewrite` checks to see whether the hostname matches the current host. If it does, the scheme and hostname are stripped out and the resulting path is treated as a URL-path. Otherwise, an external redirect is performed for the given URL. To force an external redirect back to the current host, see the `[R]` flag below.

5.3.4 - (dash)

A dash indicates that no substitution should be performed (the existing path is passed through untouched). This is used when a flag (see below) needs to be applied without changing the path.

For example, to set an environment variable without rewriting the URL:

```
RewriteRule ^/secret - [E=NEED_AUTH:1]
```

5.4 Backreferences

If the Pattern section was the “input” side of `RewriteRule`, backreferences are where things get interesting on the “output” side. Any parenthesized group in the pattern creates a backreference that you can use in the target string. These are numbered `$1` through `$9`, left to right, by opening parenthesis.

```
# Request: /products/widgets/42
RewriteRule ^/products/([^/]+)/([0-9]+)$ /catalog.php?category=$1&id=$2 [L]
# Result: /catalog.php?category=widgets&id=42
```

Here, `$1` captures `widgets` and `$2` captures `42`. The numbering follows the same rules as PCRE backreferences, which I covered in *Regular Expressions*.

You can also use backreferences from `RewriteCond` patterns in the target. These use the `%N` syntax (`%1` through `%9`) rather than `$N`, which helps you tell at a glance which part of the rule generated a particular capture:

```
RewriteCond %{HTTP_HOST} ^([^.]+\.)\.example\.com$
RewriteRule ^/(.*)$ /sites/%1/$1 [L]
```

In that example, `%1` is the subdomain captured by the `RewriteCond`, and `$1` is the path captured by the `RewriteRule`. A request for `http://blog.example.com/hello` becomes `/sites/blog/hello`.

Roy Fielding's original design for HTTP kept the URL opaque to the server — just a string to be resolved. `mod_rewrite` cheerfully violates that principle by tearing URLs apart and reassembling them from captured pieces. It's tremendously useful, but do keep in mind that you're working against the grain of the protocol's architecture every time you do it.

5.4.1 Server variables in the target

In addition to backreferences, the target string can contain server variables using the `%{VARIABLE}` syntax — the same variables available in `RewriteCond` (see *RewriteCond*).

```
# Redirect HTTP to HTTPS, preserving the host and path
RewriteCond %{HTTPS} off
RewriteRule ^(.*)$ https://%{HTTP_HOST}/$1 [R=301,L]
```

Common variables you'll use in targets include `%{HTTP_HOST}`, `%{SERVER_PORT}`, `%{REQUEST_URI}`, and `%{QUERY_STRING}`.

You can also reference `RewriteMap` functions in the target using the `${mapname:key|default}` syntax. I'll cover that in detail in *RewriteMap*.

The order in which these are expanded matters: backreferences (`$N` and `%N`) are expanded first, then server variables (`%{VARIABLE}`), then map function calls (`${mapname:...}`). In practice this means you can use a backreference *inside* a map lookup key, which is exactly how dynamic `RewriteMap`-based routing works.

5.5 Query string handling

This trips up almost everyone the first time: **the query string is not part of the pattern match**. If a user requests `/search?q=kittens`, the pattern only sees `/search`. The query string passes through to the rewritten URL unchanged, silently, behind your back.

That's usually what you want. But when it isn't, here's how to take control:

Replacing the query string — put a `?` in the target. Everything after it becomes the new query string, and the old one is discarded:

```
# /old-search?q=kittens → /new-search?type=cat
# (the original ?q=kittens is thrown away)
RewriteRule ^/old-search$ /new-search?type=cat [L]
```

Erasing the query string — end the target with a bare `?`:

```
# /page?tracking=utm_garbage → /page (clean)
RewriteRule ^/page$ /page? [L]
```

Appending to the existing query string — use the `[QSA]` (Query String Append) flag:

```
# /products/widgets → /catalog.php?category=widgets&q=kittens
# (preserves the original query string)
RewriteRule ^/products/(.+)$ /catalog.php?category=$1 [QSA,L]
```

Discarding the query string explicitly — use the `[QSD]` flag (available since `httpd 2.4.0`):

```
RewriteRule ^/clean-path$ /target [QSD,L]
```

There's also [QSL] (Query String Last), which changes how `mod_rewrite` identifies the split between the path and the query string when the target itself contains a literal `?`. See *RewriteRule Flags* for the full details on all of these.

5.6 Per-directory context gotchas

I mentioned earlier that in per-directory context (`<Directory>` blocks and `.htaccess` files), the directory prefix is stripped before matching. Let me be more specific, because this is where I see the most head-scratching on Stack Overflow and the `httpd` users mailing list.

The stripped prefix always ends with a slash. So if your rules live in `/var/www/html/.htaccess` and someone requests `/images/logo.png`, the pattern sees `images/logo.png` — no leading slash. This means a pattern that starts with `^/` will **never** match in per-directory context:

```
# In .htaccess - this NEVER matches
RewriteRule ^/images/(.*)$ /img/$1 [L]

# This is what you want
RewriteRule ^images/(.*)$ /img/$1 [L]
```

If you need to match against the full original URL-path from within a `.htaccess` file, use a `RewriteCond` with `%{REQUEST_URI}`:

```
RewriteCond %{REQUEST_URI} ^/images/(.*)$
RewriteRule ^ /img/%1 [L]
```

One more thing: although `RewriteRule` is syntactically valid inside `<Location>`, `<Files>`, and `<If>` blocks, this is unsupported and you should not do it. Relative substitutions in particular will break in creative and frustrating ways. Stick to `<Directory>`, `<VirtualHost>`, server config, and `.htaccess`.

Warning

`<If>` silently switches to per-directory context

Placing a `RewriteRule` inside an `<If>` block — even when that `<If>` is nested inside a `<VirtualHost>` — silently switches the rule to *per-directory context* behavior. This means:

- The leading slash is stripped from the URL before pattern matching.
- Substitutions trigger an internal redirect and re-entry (loop risk).
- [L] no longer truly stops processing — you need [END].

The same applies to `<Location>` and `<Files>` blocks. If your rules are doing nothing or looping unexpectedly, check whether they're wrapped in one of these containers.

Prefer placing rewrite rules directly in the `<VirtualHost>` or server-level context where they run in the URL-to-filename translation phase, with full URL-path matching and no re-entry behavior.

5.7 Home directory expansion

Here's an obscure one that has bitten a few people: when the target string begins with something that looks like `/~user` (whether from literal text or from a backreference), `mod_rewrite` performs home directory expansion automatically — even if `mod_userdir` is not loaded or configured. This happens because the expansion is built into `mod_rewrite` itself.

If this behavior surprises you (and it will, the first time it bites), you can suppress it with the `[PT]` (passthrough) flag, which hands the rewritten URL back to the normal URL mapping pipeline rather than letting `mod_rewrite` resolve it directly.

5.8 How rules are processed

RewriteRules in a given context are processed in order, top to bottom. Each rule's pattern is matched against the result of the *previous* matching rule — not against the original request. This is important:

```
RewriteRule ^/dogs/(.*)$ /pets/$1 [L]
RewriteRule ^/pets/(.*)$ /animals/$1 [L]
```

A request for `/dogs/fido` matches the first rule and is rewritten to `/pets/fido`. But the `[L]` flag stops processing, so the second rule never fires. Without the `[L]`, the second rule *would* match the output of the first — `/pets/fido` — and rewrite it to `/animals/fido`. This cascading behavior is powerful but can create unintentional loops if you're not careful. See [RewriteRule Flags](#) for more on `[L]`, `[END]`, and other flags that control the processing flow.

When `RewriteCond` directives precede a rule, the engine evaluates them only after the pattern matches — despite the fact that they appear *before* the rule in the config file. If any condition fails, the rule is skipped entirely. This is covered in detail in [RewriteCond](#).

5.9 Flags at a glance

Flags are the third argument to `RewriteRule` and modify its behavior in various ways. I cover each flag in detail in [RewriteRule Flags](#), but here's a quick reference so you can orient yourself:

Table 1: RewriteRule flag summary

Flag	Purpose
B	Escape backreferences before applying them
C	Chain this rule to the next rule
CO	Set a cookie
DPI	Discard path info
E	Set an environment variable
END	Stop processing and don't re-run in per-directory context
F	Return 403 Forbidden
G	Return 410 Gone
H	Force a content handler
L	Last rule — stop processing this ruleset
N	Re-run from the top (next round)
NC	Case-insensitive match
NE	Don't escape special characters in the output
NS	Skip if this is an internal sub-request
P	Proxy the request
PT	Pass through to the next URL mapping handler
QSA	Append the original query string
QSD	Discard the original query string
QSL	Use the last ? as the query string delimiter
R	External redirect (optionally with status code)
S	Skip the next N rules
T	Set the MIME type

5.10 Security Considerations

RewriteRule is a powerful URL manipulation tool, and with that power comes the potential for security mistakes. The following pitfalls are worth keeping in mind whenever you write rules that incorporate user-controlled input — backreferences from the URL, query string values, or HTTP headers.

5.10.1 Open Redirects

If a RewriteRule constructs a redirect URL using unvalidated user input, an attacker can craft a link that redirects visitors to a malicious site while appearing to originate from your domain. This is known as an *open redirect* vulnerability.

```
# DANGEROUS - allows open redirect
RewriteRule ^/redirect %{QUERY_STRING} [R,L]
```

An attacker could use `https://yoursite.com/redirect?https://evil.com` to redirect users to a malicious site with your domain in the address bar during the click.

Mitigation: Always validate or constrain redirect targets. If the destination must be on your own site, ensure the substitution begins with `/` (a relative path) rather than allowing a full URL from user input. Or validate against a whitelist of allowed domains.

5.10.2 Server-Side Request Forgery (SSRF)

When using the [P] (proxy) flag, `mod_rewrite` causes the server to make an HTTP request to the substitution URL on behalf of the client. If any part of that URL is derived from user input — backreferences, query strings, or headers — an attacker may be able to cause your server to make requests to arbitrary internal services or external hosts.

```
# DANGEROUS - user controls the proxy target
RewriteCond %{QUERY_STRING} target=(.+)
RewriteRule ^/fetch http://%1 [P]
```

An attacker could use this to probe internal network services (`http://169.254.169.254/latest/meta-data/` on EC2, for instance) that are not accessible from the internet.

Mitigation: Always use a fixed hostname in proxy targets. Limit backreferences to the path component only, and validate that captured values cannot contain `://` or other scheme indicators.

5.10.3 Path Traversal

Rules that map user-supplied path components directly to the filesystem can allow path traversal attacks if the input is not properly constrained:

```
# DANGEROUS - allows path traversal
RewriteRule ^/files/(.+) /var/data/$1 [L]
```

A request for `/files/../../../../etc/passwd` could potentially access files outside the intended directory (although Apache's `<Directory>` restrictions and `Options` settings provide defense in depth).

Mitigation: Use restrictive patterns — `[a-zA-Z0-9_.-]+` instead of `.+` — and rely on Apache's built-in protections as additional layers. Never assume the regex alone will prevent abuse.

5.10.4 General Principles

- **Treat backreferences as untrusted input.** Anything captured from the URL (`$1`, `$2`, ...) or from `RewriteCond` (`%1`, `%2`, ...) is user-controlled.
- **Prefer relative paths in substitutions.** A substitution starting with `/` stays on your server. A substitution that could be manipulated into `http://...` becomes a redirect or proxy to an attacker's host.
- **Use the [B] flag** when passing backreferences into query strings to prevent injection of additional parameters.
- **Least privilege:** Don't use [P] when [PT] or a simple internal rewrite suffices. Don't expose more of the filesystem than necessary.

REWRITE LOGGING

Oh, I have slipped the surly bonds of earth
And danced the skies on laughter-silvered wings.

—John Gillespie Magee Jr., *High Flight*

I can't overstate how important the rewrite log is. When your `RewriteRule` doesn't do what you expected — and it won't, at least the first three times — the rewrite log is where you go to find out what actually happened. It's the difference between debugging and guessing.

Rewrite logging uses the `LogLevel` directive with per-module trace levels. If you remember nothing else from this chapter, remember this one line:

```
LogLevel warn rewrite:trace3
```

That turns on enough `mod_rewrite` logging to see what's happening without drowning in noise. The rewrite log entries show up in your main error log (the file specified by `ErrorLog`), tagged with `[rewrite:traceN]` so you can find them.

6.1 Trace Levels

The `LogLevel` directive accepts trace levels from `trace1` through `trace8` for `mod_rewrite`. Each level includes everything from the levels above it, so `trace3` gives you `trace1` and `trace2` output as well. Here's what each level gives you:

Level	What it logs
<code>trace1</code>	Rule match/no-match results — the high-level outcome
<code>trace2</code>	Rewrite results and pass-through decisions — what the URL was rewritten to
<code>trace3</code>	Rule pattern application — which pattern was applied to which URI
<code>trace4</code>	<code>RewriteCond</code> evaluation details — the input string, the pattern, and whether it matched
<code>trace5</code>	<code>RewriteMap</code> lookups — map name, lookup key, and result (or failure)
<code>trace6</code>	Map cache behavior — cache hits and misses
<code>trace7</code>	Large data dumps (rarely useful)
<code>trace8</code>	Even larger data dumps (almost never useful)

In practice, `trace3` is the sweet spot for most debugging. It shows you which rule is being tried against which URI, without burying you in condition details. Step up to `trace4` or `trace5` when you need to understand

why a condition didn't match, or when a RewriteMap lookup is returning something unexpected.

Warning

Running at `trace6` or higher on a production server will slow things down noticeably. The server has to write a log entry for practically every internal operation `mod_rewrite` performs. Use high trace levels only for debugging, and turn them back down when you're done.

6.2 Enabling Rewrite Logging

The basic form is simple:

```
LogLevel warn rewrite:trace3
```

This sets the general log level to `warn` (so you're not flooded with info-level messages from every module) and then cranks `mod_rewrite` up to `trace3`. Rewrite log entries show up in your error log, tagged so they're easy to find.

To view just the rewrite entries in real time, filter the error log:

```
tail -f /var/log/httpd/error_log | fgrep '[rewrite:]'
```

6.3 Per-Directory Logging

Here's something I wish I'd known years earlier: since `httpd 2.3.6`, you can set `LogLevel` inside a `<Directory>`, `<Location>`, or `<VirtualHost>` block. This means you can turn on rewrite tracing for *one specific path* without flooding the log with trace output from every request to the entire server.

```
# Only debug rewrites under /api/  
<Location "/api/">  
    LogLevel warn rewrite:trace4  
</Location>
```

Or scope it to a single virtual host:

```
<VirtualHost *:443>  
    ServerName staging.example.com  
    LogLevel warn rewrite:trace3  
    # ... your rewrite rules ...  
</VirtualHost>
```

This is enormously useful on a busy server where a global `trace3` would produce an unreadable flood of log entries. Narrow the scope to the path or vhost you're actually debugging, fix the problem, and remove the directive. I cannot stress enough how much easier this makes life.

Note

Per-directory log level changes only affect messages generated *after* the request has been parsed and associated with a directory. Very early request-processing messages (connection-level events) are still controlled by the server-level LogLevel.

6.4 What's in the Rewrite Log? — An Example

The best way to talk about what's in the rewrite log is to show you some examples of the kinds of things that mod_rewrite logs.

Consider a simple rewrite scenario such as follows:

```
RewriteEngine On
RewriteCond %{REQUEST_URI} !index.php
RewriteRule . /index.php [PT,L]

LogLevel warn rewrite:trace6
```

This ruleset says “If it's not already index.php, rewrite it to index.php.”

Now, I'll make a request for the URL <http://localhost/example> and see what gets logged:

```
[Thu Sep 10 20:22:13.363463 2026] [rewrite:trace2] [pid 11879] mod_rewrite.
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] init rewrite engine with requested uri /example

[Thu Sep 10 20:22:13.363510 2026] [rewrite:trace3] [pid 11879] mod_rewrite.
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] applying pattern '.' to uri '/example'

[Thu Sep 10 20:22:13.363525 2026] [rewrite:trace4] [pid 11879] mod_rewrite.
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] RewriteCond: input='/example' pattern='!index.php' =>↪
↪matched

[Thu Sep 10 20:22:13.363533 2026] [rewrite:trace2] [pid 11879] mod_rewrite.
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] rewrite '/example' -> 'index.php'

[Thu Sep 10 20:22:13.363542 2026] [rewrite:trace2] [pid 11879] mod_rewrite.
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] local path result: index.php

[Thu Sep 10 20:22:13.575877 2026] [rewrite:trace2] [pid 11881] mod_rewrite.
↪c(468): [client 127.0.0.1:56624] 127.0.0.1 - - [localhost/sid#7f985f445348][rid
↪#7f985f949040/initial] init rewrite engine with requested uri /favicon.ico
```

(continues on next page)

(continued from previous page)

```
[Thu Sep 10 20:22:13.575920 2026] [rewrite:trace3] [pid 11881] mod_rewrite.  
↪c(468): [client 127.0.0.1:56624] 127.0.0.1 - - [localhost/sid#7f985f445348][rid  
↪#7f985f949040/initial] applying pattern '.' to uri '/favicon.ico'  
  
[Thu Sep 10 20:22:13.575935 2026] [rewrite:trace4] [pid 11881] mod_rewrite.  
↪c(468): [client 127.0.0.1:56624] 127.0.0.1 - - [localhost/sid#7f985f445348][rid  
↪#7f985f949040/initial] RewriteCond: input='/favicon.ico' pattern='!index.php' ↪  
↪=> matched  
  
[Thu Sep 10 20:22:13.575943 2026] [rewrite:trace2] [pid 11881] mod_rewrite.  
↪c(468): [client 127.0.0.1:56624] 127.0.0.1 - - [localhost/sid#7f985f445348][rid  
↪#7f985f949040/initial] rewrite '/favicon.ico' -> 'index.php'  
  
[Thu Sep 10 20:22:13.575955 2026] [rewrite:trace2] [pid 11881] mod_rewrite.  
↪c(468): [client 127.0.0.1:56624] 127.0.0.1 - - [localhost/sid#7f985f445348][rid  
↪#7f985f949040/initial] local path result: index.php
```

Let's look at the first log entry in detail:

```
[Thu Sep 10 20:22:13.363463 2026] [rewrite:trace2] [pid 11879] mod_rewrite.  
↪c(468): [client 127.0.0.1:56623] 127.0.0.1 - - [localhost/sid#7f985f445348][rid  
↪#7f985f949040/initial] init rewrite engine with requested uri /example
```

That's a lot to process all at once, so I'll break it down one field at a time.

[Thu Sep 10 20:22:13.363463 2026]

The date and time when the event occurred.

[rewrite:trace2]

The name of the module logging, and the trace level at which it is logging.

[pid 11879]

The process id of the httpd process handling this request. This will be the same across a given request. Note that in this example there are two separate requests being handled, as you'll see in a moment.

mod_rewrite.c(468):

For in-depth debugging, this is the line number in the module source code which is handling the current rewrite.

[client 127.0.0.1:56623]

The client IP address, and TCP port number on which the request connection was made.

-

This field contains the client's username in the event that the request was authenticated. In this example the request was not authenticated, so a blank value is logged.

-

In the event that the request sent ident information, this will be logged here. This hardly ever happens, and so this field will almost always be -.

```
[localhost/sid#7f985f445348][rid#7f985f949040/initial]
```

This is the unique identifier for the request.

```
init rewrite engine with requested uri /example
```

Ahah! Finally! The actual log message from mod_rewrite!

Now that you know what all of the various fields are in the log entry, let's just look at the ones I actually care about. Here's the log file again, with a lot of the superfluous information removed:

```
init rewrite engine with requested uri /example
applying pattern '.' to uri '/example'
RewriteCond: input='/example' pattern='!index.php' => matched
rewrite '/example' -> 'index.php'
local path result: index.php

init rewrite engine with requested uri /favicon.ico
applying pattern '.' to uri '/favicon.ico'
RewriteCond: input='/favicon.ico' pattern='!index.php' => matched
rewrite '/favicon.ico' -> 'index.php'
local path result: index.php
```

I've removed the extraneous information, and split the log entries into two logical chunks.

In the first bit, the requested URL `/example` is run through the ruleset and ends up getting rewritten to `/index.php`, as desired.

In the second bit, the browser requests the URL `/favicon.ico` as a side effect of the initial request. `favicon` is the icon that appears in your browser address bar next to the URL, and is an automatic feature of most browsers. As such, you're likely to see mention of `favicon.ico` in your log files from time to time, and it's nothing to worry too much about. You can read more about favicons at <http://en.wikipedia.org/wiki/Favicon>.

Follow through the log lines for the first of the two requests.

First, the rewrite engine is made aware that it needs to consider a URL, and the `init rewrite engine` log entry is made.

Next, the `RewriteRule` pattern `.` is applied to the requested URI `/example`, and this comparison is logged. In your configuration file, the `RewriteRule` appears after the `RewriteCond`, but at request time, the `RewriteRule` pattern is applied first.

Since the pattern does match, in this case, we continue to the `RewriteCond`, and the pattern `!index.php` is applied to the string `/example`. Both the pattern and the string it is being applied to are logged, which can be very useful later on in debugging rules that aren't behaving quite as you intended. This log line also tells you that the pattern matched.

Since the `RewriteRule` pattern and the `RewriteCond` both matched, we continue on to the right hand side of the `RewriteRule` and apply the rewrite, and `/example` is rewritten to `index.php`, which is also logged. A final log entry tells us what the local path result ends up being after this process, which is `index.php`.

This kind of detailed log trail tells you very specifically what's going on, and what happened at each step.

6.5 RewriteRules in .htaccess Files — An Example

I've previously discussed using `mod_rewrite` in `.htaccess` files, but it's time to see what this actually looks like in practice. Let's replace the configuration file entry above with a `.htaccess` file instead, placed in the root document directory of the website. So, I'm going to comment out several lines in the server configuration:

```
# RewriteEngine On
# RewriteCond %{REQUEST_URI} !index.php
# RewriteRule . /index.php [PT,L]

LogLevel warn rewrite:trace6
```

And instead, I'm going to place the following `.htaccess` file:

```
RewriteEngine On
RewriteCond %{REQUEST_URI} !index.php
RewriteRule . /index.php [PT,L]
```

Now, see what the log file looks like.

For the sake of brevity, let's look at just the actual log messages, and ignore all of the extra information:

```
[perdir /var/www/html/] strip per-dir prefix: /var/www/html/example -> example
[perdir /var/www/html/] applying pattern '.' to uri 'example'
[perdir /var/www/html/] input='/example' pattern='!index.php' => matched
[perdir /var/www/html/] rewrite 'example' -> '/index.php'
[perdir /var/www/html/] forcing '/index.php' to get passed through to next API_
↳URI-to-filename handler
[perdir /var/www/html/] internal redirect with /index.php [INTERNAL REDIRECT]
[perdir /var/www/html/] strip per-dir prefix: /var/www/html/index.php -> index.
↳php
[perdir /var/www/html/] applying pattern '.' to uri 'index.php'
[perdir /var/www/html/] RewriteCond: input='/index.php' pattern='!index.php' =>_
↳not-matched
[perdir /var/www/html/] pass through /var/www/html/index.php
```

The first thing you'll notice is that this is much longer than what I had before. Running rewrite rules in `.htaccess` files generally takes several more steps than when the rules are in the server configuration file, which is one of several reasons that using `.htaccess` files is so much less efficient (i.e., slower) than using the server configuration file.

Whenever possible, you should use the server configuration file rather than `.htaccess` files. (There are other reasons for this, too.)

Next, you'll notice that each log entry contains the preface:

```
[perdir /var/www/html]
```

`perdir` refers to rewrite directives that occur in per-directory context — i.e., `.htaccess` files or

<Directory> blocks. They are treated specially in a few different ways, as we'll see.

The first of these is shown in the first log entry:

```
strip per-dir prefix: /var/www/html/example -> example
```

What that means is that in per-directory context, the directory path is removed from any string before they are considered in the pattern match. Thus, rather than considering the string `/example`, as I did the first time through, now we're looking at the string `example`. This distinction matters more than it looks like it should — as we proceed to more complex examples, that leading slash will be the difference between a pattern matching and not matching, so you need to be aware of this every time you use `.htaccess` files.

The next few lines of the log proceed as before, except that we're looking at `example` rather than `/example` in each line. Carefully compare the log entries from the first time through to the ones this time.

What happens next is a surprise to most first-time users of `mod_rewrite`. The requested URI `example` is redirected to the URI `/index.php`, and the whole process starts over again with that new URL. This is because, in per-directory context, once a rewrite has been executed, that target URL must get passed back to the URL mapping process to determine what that URL maps to ... which may include invoking a `.htaccess` file.

In this case, this causes the ruleset to be executed all over again, with the rewritten URL `/index.php`.

The remainder of the log should look very familiar. It's the same as what we saw before, with `/index.php` getting stripped to `index.php` and run through the paces. This time around, however, the `RewriteCond` does not match, and so the request is passed through unchanged.

6.6 Debugging RewriteMap Lookups

If you're using a `RewriteMap` (see *RewriteMap*) and the lookup isn't returning what you expect, `trace5` is where you want to be. At that level, `mod_rewrite` logs every map lookup — the map name, the key that was looked up, and the result (or the fact that the lookup failed).

```
LogLevel warn rewrite:trace5
```

You'll see log entries like:

```
map lookup OK: map=examplemap key=foo -> val=bar
map lookup FAILED: map=examplemap key=baz
```

This is invaluable when you're debugging `txt` or `dbm` maps where a typo in the map file (an extra space, a missing entry) can silently cause a lookup to fail and your rule to not match. At `trace6`, you'll also cache-related entries — whether the map result came from the internal cache or required a fresh lookup. This is mostly useful if you suspect the map file has been updated but the cached values are stale.

6.7 Common Error Messages

AH00124: Request exceeded the limit of 10 internal redirects

If you see this in your error log:

```
AH00124: Request exceeded the limit of 10 internal redirects due to
probable configuration error. Use 'LimitInternalRecursion' to increase
the limit if necessary. Use 'LogLevel debug' to get a backtrace.
```

your rewrite rules are looping. In per-directory context, each successful substitution triggers an internal redirect that re-runs the ruleset from the top. If nothing stops the cycle, Apache hits the default limit of 10 and returns a 500 error.

The fix is almost always one of:

- Replace [L] with [END] (see *END*).
- Add a RewriteCond that prevents re-matching (e.g., `#{REQUEST_FILENAME} !-f`).
- Check `#{THE_REQUEST}` which is never modified by rewrites.

Do **not** increase `LimitInternalRecursion` to paper over a loop — fix the underlying rule logic.

The phantom 301: browser caching

A notoriously frustrating debugging scenario: you fix a broken 301 redirect, reload, and the browser still goes to the old destination. This happens because browsers cache 301 (permanent) redirects indefinitely — there is no expiry unless you set one.

Tips:

- Use `[R=302]` (temporary) during development and testing. Switch to `[R=301]` only when you're confident the rule is correct.
- When debugging a stuck 301: clear the browser cache, use an incognito/private window, or test with `curl -v` (which never caches).
- If you've already shipped a bad 301 to real users, you may need to put a corrective 301 at the *old wrong destination* pointing to the right place, since their browsers cached the first redirect.

Other AH error codes

Apache httpd uses thousands of `AH#####` error codes across all its modules. For a searchable reference covering every code — with explanations and fix suggestions — see <https://httpd.rcbowen.com/errors/>

6.8 Don't Leave It On

I want to re-emphasize: don't leave trace-level rewrite logging enabled on a production server. Every trace-level log entry requires `mod_rewrite` to format a string, acquire a lock on the log file, write the entry, and release the lock — for *every single request* that touches a rewrite rule.

On a server handling thousands of requests per second, `rewrite:trace6` can measurably increase response times and generate gigabytes of log data in short order. I've seen it fill a disk partition in under an hour on a busy server.

The workflow is: turn it on, reproduce the problem, read the log, turn it off. If you're using per-directory logging (see *Per-Directory Logging* above), the blast radius is already limited — but even so, clean up after yourself.

REWRITERULE FLAGS

My girl came to the study
and said Help me;
I told her I had a time problem
which meant:
I would die for you but I don't have ten minutes.

—Brenda Hillman, *Time Problem*

Flags modify the behavior of the rule. You may have zero or more flags, and the effect is cumulative. Flags may be repeated where appropriate. For example, you may set several environment variables by using several [E] flags, or set several cookies with multiple [CO] flags. Flags are separated with commas:

```
[B,C,NC,PT,L]
```

There are a *lot* of flags. Here they are:

7.1 B - escape backreferences

The [B] flag instructs RewriteRule to escape non-alphanumeric characters before applying the transformation. `mod_rewrite` has to unescape URLs before mapping them, so backreferences are unescaped at the time they are applied. Using the B flag, non-alphanumeric characters in backreferences will be escaped. (See backreferences for discussion of backreferences.) For example, consider the rule:

```
RewriteRule ^search/(.*)$ /search.php?term=$1
```

Given a search term of 'x & y/z', a browser will encode it as 'x%20%26%20y%2Fz', making the request 'search/x%20%26%20y%2Fz'. Without the B flag, this rewrite rule will map to 'search.php?term=x & y/z', which isn't a valid URL, and so would be encoded as `search.php?term=x%20&y%2Fz=`, which is not what was intended.

With the B flag set on this same rule, the parameters are re-encoded before being passed on to the output URL, resulting in a correct mapping to `/search.php?term=x%20%26%20y%2Fz`.

Note that you may also need to set `AllowEncodedSlashes` to `On` to get this particular example to work, as `httpd` does not allow encoded slashes in URLs, and returns a 404 if it sees one.

This escaping is particularly necessary in a proxy situation, when the backend may break if presented with an unescaped URL.

Note

Starting in version 2.4.26, you can limit which characters are escaped by the [B] flag by providing a list of characters as an argument. For example:

```
[B=#?;]
```

This limits escaping to only the #, ?, and ; characters. If you need to include a space in the list of characters to escape, you must quote the entire third argument of the RewriteRule:

```
RewriteRule "^search/(.*)$" "/search.php?term=$1" "[B= #?;]"
```

7.2 BNP - backrefnoplus

The [BNP] flag, short for “backrefnoplus,” modifies the behavior of the [B] flag so that spaces in backreferences are escaped to %20 rather than +. This is important when the backreference is used in the path component of the URL rather than in the query string, since + is only interpreted as a space in query strings, not in paths.

Use this flag in conjunction with the [B] flag when the substitution places the backreference in the path portion of the URL.

```
RewriteRule "^search/(.*)$" "/search.php/$1" "[B,BNP]"
```

In this example, a search term containing spaces will be escaped as %20 in the resulting path, rather than being converted to + signs.

2.4.26 This flag is available in httpd 2.4.26 and later.

7.3 BCTLS

The [BCTLS] flag works like the [B] flag, but only escapes control characters (bytes 0x00 through 0x1F and 0x7F) and the space character (0x20). This is the same set of characters that would be rejected if they were copied unencoded into a query string.

Use this flag when you need a lighter-weight escaping than the full [B] flag provides — for example, when your backreferences may contain spaces or control characters but you want to preserve other special characters as-is.

```
RewriteRule "^search/(.*)$" "/search.php/$1" "[BCTLS]"
```

trunk This flag is only available in httpd trunk (future 2.5.x) and is not yet in any stable release.

7.4 BNE

The [BNE] flag provides an exclusion list for the [B] or [BCTLS] flags. Characters listed in the [BNE] argument will *not* be escaped by the backreference escaping process. This is useful when certain characters in backreferences are meaningful and should be preserved as-is.

The syntax is [BNE=<characters>], where <characters> is the set of characters to exclude from escaping. For example, to exclude forward slashes from being escaped:

```
RewriteRule "^search/(.*)$" "/search.php?term=$1" "[B,BNE=/"
```

In this example, the [B] flag will escape all non-alphanumeric characters in the backreference *except* the forward slash, which will be passed through unchanged.

trunk This flag is only available in httpd trunk (future 2.5.x) and is not yet in any stable release.

7.5 C - chain

The [C] or [chain] flag indicates that the RewriteRule is chained to the next rule. That is, if the rule matches, then it is processed as usual and control moves on to the next rule. However, if it does not match, then the next rule, and any other rules that are chained together, will be skipped.

7.6 CO - cookie

The [CO], or [cookie] flag, allows you to set a cookie when a particular RewriteRule matches. The argument consists of three required fields and four optional fields.

The full syntax for the flag, including all attributes, is as follows:

```
[CO=NAME:VALUE:DOMAIN:lifetime:path:secure:httponly]
```

You must declare a name, a value, and a domain for the cookie to be set.

7.6.1 Domain

The domain for which you want the cookie to be valid. This may be a hostname, such as `www.example.com`, or it may be a domain, such as `.example.com`. It must be at least two parts separated by a dot. That is, it may not be merely `.com` or `.net`. Cookies of that kind are forbidden by the cookie security model. You may optionally also set the following values:

7.6.2 Lifetime

The time for which the cookie will persist, in minutes. A value of 0 indicates that the cookie will persist only for the current browser session. This is the default value if none is specified.

7.6.3 Path

The path, on the current website, for which the cookie is valid, such as `/customers/` or `/files/download/`. By default, this is set to `/` - that is, the entire website.

7.6.4 Secure

If set to `secure`, `true`, or `1`, the cookie will only be permitted to be translated via secure (https) connections.

7.6.5 httponly

If set to `HttpOnly`, `true`, or `1`, the cookie will have the `HttpOnly` flag set, which means that the cookie will be inaccessible to JavaScript code on browsers that support this feature.

7.6.6 Example

Consider this example:

```
RewriteEngine On
RewriteRule ^/index\.html - [CO=frontdoor:yes:.example.com:1440:/]
```

In the example given, the rule doesn't rewrite the request. The `'-'` rewrite target tells `mod_rewrite` to pass the request through unchanged. Instead, it sets a cookie called `'frontdoor'` to a value of `'yes'`. The cookie is valid for any host in the `.example.com` domain. It will be set to expire in 1440 minutes (24 hours) and will be returned for all URIs (i.e., for the path `'/'`).

7.7 DPI - discardpath

The `DPI` flag causes the `PATH_INFO` portion of the rewritten URI to be discarded.

This flag is available in version 2.2.12 and later.

In per-directory context, the URI each `RewriteRule` compares against is the concatenation of the current values of the URI and `PATH_INFO`.

The current URI can be the initial URI as requested by the client, the result of a previous round of `mod_rewrite` processing, or the result of a prior rule in the current round of `mod_rewrite` processing.

In contrast, the `PATH_INFO` that is appended to the URI before each rule reflects only the value of `PATH_INFO` before this round of `mod_rewrite` processing. As a consequence, if large portions of the URI are matched and copied into a substitution in multiple `RewriteRule` directives, without regard for which parts of the URI came from the current `PATH_INFO`, the final URI may have multiple copies of `PATH_INFO` appended to it.

Use this flag on any substitution where the `PATH_INFO` that resulted from the previous mapping of this request to the filesystem is not of interest. This flag permanently forgets the `PATH_INFO` established before this round of `mod_rewrite` processing began. `PATH_INFO` will not be recalculated until the current round of `mod_rewrite` processing completes. Subsequent rules during this round of processing will see only the direct result of substitutions, without any `PATH_INFO` appended.

7.8 E - env

With the [E], or [env] flag, you can set the value of an environment variable. Note that some environment variables may be set after the rule is run, thus unsetting what you have set.

The full syntax for this flag is:

```
[E=VAR:VAL]
[E=!VAR]
```

VAL may contain backreferences (See section backreferences) (\$N or %N) which will be expanded.

Using the short form

```
[E=VAR]
```

you can set the environment variable named VAR to an empty value.

The form

```
[E=!VAR]
```

allows you to unset a previously set environment variable named VAR.

Environment variables can then be used in a variety of contexts, including CGI programs, other RewriteRule directives, or CustomLog directives.

The following example sets an environment variable called ‘image’ to a value of ‘1’ if the requested URI is an image file. Then, that environment variable is used to exclude those requests from the access log.

```
RewriteRule \.(png|gif|jpg)$ - [E=image:1]
CustomLog logs/access_log combined env=!image
```

Note that this same effect can be obtained using SetEnvIf. This technique is offered as an example, not as a recommendation.

The [E] flag may be repeated if you want to set more than one environment variable at the same time:

```
RewriteRule \.pdf$ [E=document:1,E=pdf:1,E=done]
```

Warning

REDIRECT_ prefix after internal redirects

In *per-directory context*, a successful substitution triggers an internal redirect. When this happens, all environment variables set during the previous pass — including those created with [E=VAR:VAL] — are renamed with a REDIRECT_ prefix. A variable you set as rewritten becomes REDIRECT_rewritten in the redirected request.

This catches many users off-guard. If you set an environment variable and then test for it later in the same ruleset, it may not exist under the name you expect after the internal redirect occurs.

To test for the renamed variable:

```
# Set on first pass
RewriteRule ^/old /new [E=was_old:1,L]

# On the re-entry pass, the variable has been renamed:
RewriteCond %{ENV:REDIRECT_was_old} =1
RewriteRule ^ - [E=confirmed:1]
```

If the request goes through multiple internal redirects, prefixes accumulate: REDIRECT_REDIRECT_was_old, and so on. In practice, testing for REDIRECT_ plus the original name covers the common case.

This behavior does **not** apply in server/virtualhost context (no internal redirect occurs there), nor does it affect external redirects — those start an entirely new request on the client side.

See also the [CO] (cookie) flag if you need state that survives across external redirects.

7.9 END

Although the flags are presented here in alphabetical order, it makes more sense to go read the section about the L flag first (*L - last*) and then come back here.

Using the [END] flag terminates not only the current round of rewrite processing (like [L]) but also prevents any subsequent rewrite processing from occurring in *per-directory context*.

This distinction matters because of how per-directory rewriting works: after a substitution, Apache re-enters the per-directory processing pipeline and the ruleset runs again from the top. The [L] flag stops the current pass but does *not* prevent re-entry. The [END] flag does — it is the cleanest way to say “this request is fully rewritten, do not re-process it.”

[END] is available since httpd 2.3.9 and is the recommended loop-prevention flag for per-directory rewrites.

This does not apply to new requests resulting from external redirects.

7.10 F - forbidden

Using the [F] flag causes the server to return a 403 Forbidden status code to the client. While the same behavior can be accomplished using the Deny directive, this allows more flexibility in assigning a Forbidden status.

The following rule will forbid .exe files from being downloaded from your server.

```
RewriteRule \.exe - [F]
```

This example uses the “-” syntax for the rewrite target, which means that the requested URI is not modified. There’s no reason to rewrite to another URI, if you’re going to forbid the request.

When using [F], an [L] is implied - that is, the response is returned immediately, and no further rules are evaluated.

7.11 G - gone

The [G] flag forces the server to return a 410 Gone status with the response. This indicates that a resource used to be available, but is no longer available.

As with the [F] flag, you will typically use the “-” syntax for the rewrite target when using the [G] flag:

```
RewriteRule oldproduct - [G,NC]
```

When using [G], an [L] is implied - that is, the response is returned immediately, and no further rules are evaluated.

7.12 H - handler

Forces the resulting request to be handled with the specified handler. For example, one might use this to force all files without a file extension to be parsed by the php handler:

```
RewriteRule !\.\. - [H=application/x-httpd-php]
```

The regular expression above - !\.\. - will match any request that does not contain the literal . character.

This can be also used to force the handler based on some conditions. For example, the following snippet used in per-server context allows .php files to be displayed by mod_php if they are requested with the .phps extension:

```
RewriteRule ^(/source/.\.\.php)s$ $1 [H=application/x-httpd-php-source]
```

The regular expression above - ^(/source/.\.\.php)s\$ - will match any request that starts with /source/ followed by 1 or n characters followed by .phps literally. The backreference \$1 refers to the captured match within parentheses of the regular expression.

7.13 L - last

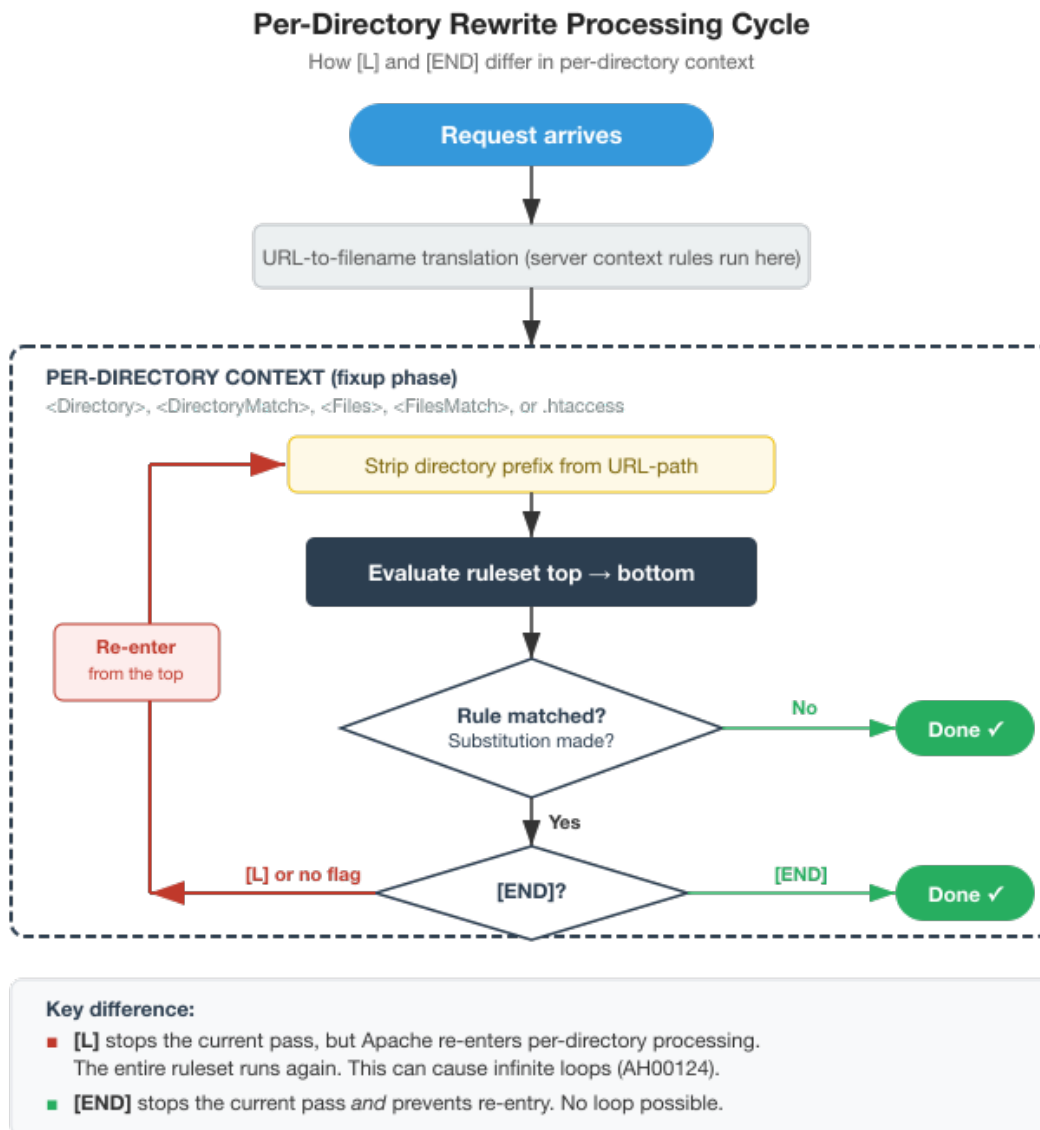
The [L] flag causes mod_rewrite to stop processing the rule set. In most contexts, this means that if the rule matches, no further rules will be processed. This corresponds to the last command in Perl, or the break command in C. Use this flag to indicate that the current rule should be applied immediately without considering further rules.

If you are using RewriteRule in either .htaccess files or in <Directory> sections — that is, in *per-directory context* — it is important to understand that [L] means “stop processing the ruleset *for this pass*,” not “stop forever.” Once the rules have been processed, the rewritten request is handed back to the URL parsing engine. As the rewritten request is handled, the same .htaccess file or <Directory> section may be encountered again, and the entire ruleset runs again from the start. This re-entry is a fundamental part of how per-directory rewriting works — it is not a bug.

The most common symptom of this behavior is an infinite loop: a rule rewrites /foo to /index.php?q=foo with [L], the request re-enters, and the rule matches index.php again, rewriting it a second time. Apache eventually hits its internal redirect limit and returns a 500 error:

AH00124: Request exceeded the limit of 10 internal redirects due to probable configuration error.

The following diagram illustrates the per-directory re-invocation cycle and shows where [L] and [END] diverge:



It is therefore important, if you are using RewriteRule directives in one of these contexts, that you take explicit steps to avoid rules looping, and not count solely on the [L] flag to terminate execution of a series of rules, as shown below.

The preferred solution is the [END] flag (available since httpd 2.3.9), which terminates not only the current round of rewrite processing but prevent any subsequent rewrite processing from occurring in per-directory context. If you know you want “stop, we’re done,” use [END] instead of [L]. [END] does not apply to new requests resulting from external redirects.

Other ways to break the loop include adding a RewriteCond that checks whether the request has already been rewritten (e.g. RewriteCond %{REQUEST_FILENAME} !=-f or RewriteCond %{THE_REQUEST} !index\.php). See *Recipes* for worked examples.

The example given here will rewrite any request to index.php, giving the original request as a query string argument to index.php, however, the RewriteCond ensures that if the request is already for index.php, the RewriteRule will be skipped.

```
RewriteBase /
RewriteCond %{REQUEST_URI} !=/index.php
RewriteRule ^(.*) /index.php?req=$1 [L,PT]
```

See the RewriteCond chapter for further discussion of the RewriteCond directive.

7.14 N - next

The [N] flag causes the ruleset to start over again from the top, using the result of the ruleset so far as a starting point. Use with extreme caution, as it may result in a loop.

The [N] flag could be used, for example, if you wished to replace a certain string or letter repeatedly in a request. The example shown here will replace A with B everywhere in a request, and will continue doing so until there are no more As to be replaced.

```
RewriteRule (.*?)A(.*?) $1B$2 [N]
```

You can think of this as a while loop: While this pattern still matches (i.e., while the URI still contains an A), perform this substitution (i.e., replace the A with a B).

7.15 NC - nocase

Use of the [NC] flag causes the RewriteRule to be matched in a case-insensitive manner. That is, it doesn't care whether letters appear as upper-case or lower-case in the matched URI.

In the example below, any request for an image file will be proxied to your dedicated image server. The match is case-insensitive, so that .jpg and .JPG files are both acceptable, for example.

```
RewriteRule (.*\.(jpg|gif|png))$ http://images.example.com$1 [P,NC]
```

7.16 NE - noescape

By default, special characters, such as & and ?, for example, will be converted to their hexcode equivalent. Using the [NE] flag prevents that from happening.

```
RewriteRule ^/anchor/(.+) /bigpage.html#$1 [NE,R]
```

The above example will redirect /anchor/xyz to /bigpage.html#xyz. Omitting the [NE] will result in the # being converted to its hexcode equivalent, %23, which will then result in a 404 Not Found error condition.

7.17 NS - nosubreq

Use of the [NS] flag prevents the rule from being used on subrequests. For example, a page which is included using an SSI (Server Side Include) is a subrequest, and you may want to avoid rewrites happening on those subrequests. Also, when `mod_dir` tries to find out information about possible directory default files (such as `index.html` files), this is an internal subrequest, and you often want to avoid rewrites on such subrequests. On subrequests, it is not always useful, and can even cause errors, if the complete set of rules are applied. Use this flag to exclude problematic rules.

To decide whether or not to use this rule: if you prefix URLs with CGI-scripts, to force them to be processed by the CGI-script, it's likely that you will run into problems (or significant overhead) on sub-requests. In these cases, use this flag.

Images, javascript files, or css files, loaded as part of an HTML page, are not subrequests - the browser requests them as separate HTTP requests.

7.18 P - proxy

Use of the [P] flag causes the request to be handled by `mod_proxy`, and handled via a proxy request. For example, if you wanted all image requests to be handled by a back-end image server, you might do something like the following:

```
RewriteRule /(.*).\.(jpg|gif|png)$ http://images.example.com/$1.$2 [P]
```

Use of the [P] flag implies [L]. That is, the request is immediately pushed through the proxy, and any following rules will not be considered.

You must make sure that the substitution string is a valid URI (typically starting with `<http://hostname>`) which can be handled by the `mod_proxy`. If not, you will get an error from the proxy module. Use this flag to achieve a more powerful implementation of the `ProxyPass` directive, to map remote content into the namespace of the local server.

Security Warning

Take care when constructing the target URL of the rule, considering the security impact from allowing the client influence over the set of URLs to which your server will act as a proxy. Ensure that the scheme and hostname part of the URL is either fixed, or does not allow the client undue influence.

Performance warning

Using this flag triggers the use of `mod_proxy`, without handling of persistent connections. This means the performance of your proxy will be better if you set it up with `ProxyPass` or `ProxyPassMatch`.

This is because this flag triggers the use of the default worker, which does not handle connection pooling. Avoid using this flag and prefer those directives, whenever you can.

Note: `mod_proxy` must be enabled in order to use this flag.

See Chapter *Proxies and mod_rewrite* for a more thorough treatment of proxying.

7.19 PT - passthrough

The target (or substitution string) in a RewriteRule is assumed to be a file path, by default. The use of the [PT] flag causes it to be treated as a URI instead. That is to say, the use of the [PT] flag causes the result of the RewriteRule to be passed back through URL mapping, so that location-based mappings, such as Alias, Redirect, or ScriptAlias, for example, might have a chance to take effect.

If, for example, you have an Alias for /icons, and have a RewriteRule pointing there, you should use the [PT] flag to ensure that the Alias is evaluated.

```
Alias /icons /usr/local/apache/icons
RewriteRule /pics/(.+)\.jpg$ /icons/$1.gif [PT]
```

Omission of the [PT] flag in this case will cause the Alias to be ignored, resulting in a 'File not found' error being returned.

The [PT] flag implies the [L] flag: rewriting will be stopped in order to pass the request to the next phase of processing.

Note that the [PT] flag is implied in per-directory contexts such as <Directory> sections or in .htaccess files. The only way to circumvent that is to rewrite to -.

7.20 QSA - qsappend

When the replacement URI contains a query string, the default behavior of RewriteRule is to discard the existing query string, and replace it with the newly generated one. Using the [QSA] flag causes the query strings to be combined.

Consider the following rule:

```
RewriteRule /pages/(.+) /page.php?page=$1 [QSA]
```

With the [QSA] flag, a request for /pages/123?one=two will be mapped to /page.php?page=123&one=two. Without the [QSA] flag, that same request will be mapped to /page.php?page=123 - that is, the existing query string will be discarded.

7.21 QSD - qsdiscard

When the requested URI contains a query string, and the target URI does not, the default behavior of RewriteRule is to copy that query string to the target URI. Using the [QSD] flag causes the query string to be discarded.

2.4 This flag is available in httpd 2.4.0 and later.

Using [QSD] and [QSA] together will result in [QSD] taking precedence.

If the target URI has a query string, the default behavior will be observed - that is, the original query string will be discarded and replaced with the query string in the RewriteRule target URI.

7.22 QSL - qslast

By default, the first ? character in the substitution string separates the path from the query string. The [QSL] flag (short for “query string last”) changes this behavior so that the *last* ? character is used as the separator instead.

This is useful when you need to map requests to files that contain literal question marks in their filenames. Without this flag, a substitution containing multiple ? characters would incorrectly split the path and query string at the first ?.

```
# Map to a file with a literal '?' in its name
RewriteRule "^test$" "/apath/file?name?q=1" [QSL]
```

In this example, the substitution is split at the *last* ?, so the path becomes /apath/file?name and the query string becomes q=1.

2.4.19 This flag is available in httpd 2.4.19 and later.

7.23 R - redirect

Use of the [R] flag causes an HTTP redirect to be issued to the browser. If a fully-qualified URL is specified (that is, including <http://servername/>) then a redirect will be issued to that location. Otherwise, the current protocol, servername, and port number will be used to generate the URL sent with the redirect.

Any valid HTTP response status code may be specified, using the syntax [R=305], with a 302 status code being used by default if none is specified. The status code specified need not necessarily be a redirect (3xx) status code. However, if a status code is outside the redirect range (300-399) then the substitution string is dropped entirely, and rewriting is stopped as if the L were used.

In addition to response status codes, you may also specify redirect status using their symbolic names: temp (default), permanent, or seeother.

You will almost always want to use [R] in conjunction with [L] (that is, use [R,L]) because on its own, the [R] flag prepends <http://thishost%5B:thisport%5D> to the URI, but then passes this on to the next rule in the ruleset, which can often result in ‘Invalid URI in request’ warnings.

7.24 S - skip

The [S] flag is used to skip rules that you don’t want to run. The syntax of the skip flag is [S=N], where N signifies the number of rules to skip (provided the RewriteRule and any preceding RewriteCond directives match). This can be thought of as a goto statement in your rewrite ruleset. In the following example, we only want to run the RewriteRule if the requested URI doesn’t correspond with an actual file.

```
# Is the request for a non-existent file?
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# If so, skip these two RewriteRules
RewriteRule .? - [S=2]
```

(continues on next page)

(continued from previous page)

```
RewriteRule (*.gif) images.php?$1
RewriteRule (*.html) docs.php?$1
```

This technique is useful because a RewriteCond only applies to the RewriteRule immediately following it. Thus, if you want to make a RewriteCond apply to several RewriteRule`s, one possible technique is to negate those conditions and add a RewriteRule with a [Skip] flag. You can 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:

```
# Does the file exist?
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# Create an if-then-else construct by skipping 3 lines if we meant to go to the
↪ "else" stanza.
RewriteRule .? - [S=3]

# IF the file exists, then:
  RewriteRule (*.gif) images.php?$1
  RewriteRule (*.html) docs.php?$1
  # Skip past the "else" stanza.
  RewriteRule .? - [S=1]
# ELSE...
  RewriteRule (*) 404.php?file=$1
# END
```

It is probably easier to accomplish this kind of configuration using the <If>, <ElseIf>, and <Else> directives instead. (2.4 and later - See <If>, <Elsif>, and <Else>.)

7.25 T - type

Sets the MIME type with which the resulting response will be sent. This has the same effect as the AddType directive.

For example, you might use the following technique to serve Perl source code as plain text, if requested in a particular way:

```
# Serve .pl files as plain text
RewriteRule \.pl$ - [T=text/plain]
```

Or, perhaps, if you have a camera that produces jpeg images without file extensions, you could force those images to be served with the correct MIME type by virtue of their file names:

```
# Files with 'IMG' in the name are jpg images.
RewriteRule IMG - [T=image/jpg]
```

Note that this particular example could be better done using `<FilesMatch>` instead. Always consider the alternate solutions to a problem before resorting to rewrite, which will invariably be a less efficient solution than the alternatives.

If used in per-directory context, use only - (dash) as the substitution for the entire round of `mod_rewrite` processing, otherwise the MIME-type set with this flag is lost due to an internal re-processing (including subsequent rounds of `mod_rewrite` processing). The `L` flag can be useful in this context to end the current round of `mod_rewrite` processing.

7.26 UnsafeAllow3F

By default, if the request URI contains an encoded question mark (`%3f`) and the substitution string contains a literal `?` (separating path from query string), `mod_rewrite` will halt the rewrite and return a 403 Forbidden response. This safeguard protects against scenarios where an attacker could manipulate captured encoded question marks to inject unexpected query string parameters.

The `[UnsafeAllow3F]` flag disables this protection and allows the rewrite to proceed even when the request URI contains `%3f` and the substitution contains a `?`.

Warning

This is a security flag. Use it with caution and only when you fully understand the implications. Enabling this flag may allow malicious clients to manipulate the boundary between the path and query string components of the rewritten URL.

```
RewriteRule "^(.*)$" "/page.php?uri=$1" [UnsafeAllow3F]
```

7.27 UnsafePrefixStat

When a `RewriteRule` is used in server (virtualhost) context and the substitution string begins with a variable or backreference that resolves to a filesystem path, `mod_rewrite` will automatically prefix the substitution with the document root. This prevents the substitution from inadvertently mapping to a path outside the intended document tree.

The `[UnsafePrefixStat]` flag disables this automatic prefixing, allowing the substitution to resolve to an absolute filesystem path derived from variables or backreferences.

Warning

This is a security flag. Without the automatic document root prefix, a rewrite rule could potentially map to files outside the document root. Use this flag only when you have verified that the substitution cannot be manipulated to access unintended filesystem locations.

```
RewriteCond "%{DOCUMENT_ROOT}" "(.+)"  
RewriteRule "^(.*)$" "%1/sub/$1" [UnsafePrefixStat]
```

Without the [UnsafePrefixStat] flag, the %1 backreference at the start of the substitution would cause mod_rewrite to prefix the result with the document root, potentially producing an incorrect double-rooted path.

trunk This flag is only available in httpd trunk (future 2.5.x) and is not yet in any stable release.

7.28 UNC

The [UNC] flag prevents mod_rewrite from merging multiple leading slashes in the substitution result into a single slash. This is necessary on Windows systems when the substitution resolves to a UNC (Universal Naming Convention) path such as \\server\share, which requires the double leading slashes to be preserved.

Note that this flag is only needed when the multiple leading slashes come from a variable or backreference expansion. If the substitution starts with multiple literal slashes, they are preserved automatically.

```
RewriteCond "%{REQUEST_URI}" "^/share/(.*)"  
RewriteRule "^(.*)$" "//fileserver/%1" [UNC]
```

trunk This flag is only available in httpd trunk (future 2.5.x) and is not yet in any stable release.

REWRITECOND

Snore on in your front row seat
Let not my voice disturb the wordless heaven your eyes have found

—James Kirkup, *To An Old Lady Asleep At A Poetry Reading*

The `RewriteCond` directive attaches additional conditions on a `RewriteRule`, and may also set backreferences that may be used in the rewrite target.

One or more `RewriteCond` directives may precede a `RewriteRule` directive. That `RewriteRule` is then applied only if the current state of the URI matches its pattern, and all of these conditions are met.

The `RewriteCond` directive has the following syntax:

```
RewriteCond TestString CondPattern [Flag]
```

The arguments have the following meaning:

TestString

Any string or variable to be tested for a match.

CondPattern

A regular expression or other expression to be compared against the `TestString`.

Flag

One or more flags which modify the behavior of the condition.

These definitions will be expanded in the sections below.

8.1 TestString

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

RewriteRule backreferences

These are backreferences of the form `$N` ($0 \leq N \leq 9$). `$1` to `$9` provide access to the grouped parts (in parentheses) of the pattern, from the `RewriteRule` which is subject to the current set of `RewriteCond` conditions. `$0` provides access to the whole string matched by that pattern.

RewriteCond backreferences

These are backreferences of the form `%N` ($0 \leq N \leq 9$). `%1` to `%9` provide access to the grouped

parts (again, in parentheses) of the pattern, from the last matched RewriteCond in the current set of conditions. %0 provides access to the whole string matched by that pattern.

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:

- HTTP_USER_AGENT
- HTTP_REFERER
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_PROXY_CONNECTION
- HTTP_ACCEPT

Connection & request:

- REMOTE_ADDR
- REMOTE_HOST
- REMOTE_PORT
- REMOTE_USER
- REMOTE_IDENT
- REQUEST_METHOD
- SCRIPT_FILENAME
- PATH_INFO
- QUERY_STRING
- AUTH_TYPE

Server internals:

- DOCUMENT_ROOT
- SERVER_ADMIN
- SERVER_NAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- SERVER_SOFTWARE

Date and time:

- TIME_YEAR
- TIME_MON
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_SEC
- TIME_WDAY
- TIME

Specials:

- API_VERSION
- THE_REQUEST
- REQUEST_URI
- REQUEST_FILENAME
- IS_SUBREQ
- HTTPS
- REQUEST_SCHEME

These variables all correspond to the similarly named HTTP MIME-headers, C variables of the Apache HTTP Server or struct tm fields of the Unix system. Most are documented elsewhere in the Manual or in the CGI specification.

SERVER_NAME and SERVER_PORT depend on the values of UseCanonicalName and UseCanonicalPhysicalPort respectively.

Those that are special to mod_rewrite include those below.

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 httpd 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 httpd in use (in the release version of Apache httpd 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. This value has not been unescaped (decoded), unlike most other variables below.

REQUEST_URI

The path component of the requested URI, such as “/index.html”. This notably excludes the query string which is available as its own variable named QUERY_STRING.

REQUEST_FILENAME

The full local filesystem path to the file or script matching the request, if this has already been determined by the server at the time REQUEST_FILENAME is referenced. Otherwise, such as when used in virtual host context, the same value as REQUEST_URI. Depending on the value of AcceptPathInfo, the server may have only used some leading components of the REQUEST_URI to map the request to a file.

HTTPS

Will contain the text “on” if the connection is using SSL/TLS, or “off” otherwise. (This variable can be safely used regardless of whether or not mod_ssl is loaded).

REQUEST_SCHEME

Will contain the scheme of the request (usually “http” or “https”). This value can be influenced with ServerName.

If the TestString has the special value expr, the CondPattern will be treated as an ap_expr. HTTP headers referenced in the expression will be added to the Vary header if the novary flag is not given.

Other things you should be aware of:

The variables SCRIPT_FILENAME and REQUEST_FILENAME contain the same value - the value of the filename field of the internal request_rec structure of the Apache HTTP Server. The first name is the commonly known CGI variable name while the second is the appropriate counterpart of REQUEST_URI (which contains the value of the uri field of request_rec).

If a substitution occurred and the rewriting continues, the value of both variables will be updated accordingly.

If used in per-server context (i.e., before the request is mapped to the filesystem) `SCRIPT_FILENAME` and `REQUEST_FILENAME` cannot contain the full local filesystem path since the path is unknown at this stage of processing. Both variables will initially contain the value of `REQUEST_URI` in that case. In order to obtain the full local filesystem path of the request in per-server context, use an URL-based look-ahead `%{LA-U:REQUEST_FILENAME}` to determine the final value of `REQUEST_FILENAME`.

`%{ENV:variable}`, where `variable` can be any environment variable, is also available. This is looked-up via internal Apache httpd structures and (if not found there) via `getenv()` from the Apache httpd server process.

`%{SSL:variable}`, where `variable` is the name of an SSL environment variable, can be used whether or not `mod_ssl` is loaded, but will always expand to the empty string if it is not. Example: `%{SSL:SSL_CIPHER_USEKEYSIZE}` may expand to 128.

`%{HTTP:header}`, where `header` can be any HTTP MIME-header name, can always be used to obtain the value of a header sent in the HTTP request. Example: `%{HTTP:Proxy-Connection}` is the value of the HTTP header `Proxy-Connection`.

If an HTTP header is used in a condition this header is added to the Vary header of the response in case the condition evaluates to true for the request. It is not added if the condition evaluates to false for the request. Adding the HTTP header to the Vary header of the response is needed for proper caching.

It has to be kept in mind that conditions follow a short circuit logic in the case of the ‘`ornext|OR`’ flag so that certain conditions might not be evaluated at all.

`%{LA-U:variable}` can be used for look-aheads which perform an internal (URL-based) sub-request to determine the final value of `variable`. This can be used to access `variable` for rewriting which is not available at the current stage, but will be set in a later phase.

For instance, to rewrite according to the `REMOTE_USER` variable from within the per-server context (`httpd.conf` file) you must use `%{LA-U:REMOTE_USER}` - this variable is set by the authorization phases, which come after the URL translation phase (during which `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}` in that context.

`%{LA-F:variable}` can be used to perform 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.

8.2 CondPattern

`CondPattern` is the condition pattern, a regular expression which is applied to the current instance of the `TestString`. `TestString` is first evaluated, before being matched against `CondPattern`.

`CondPattern` is usually a perl compatible regular expression, but there is additional syntax available to perform other useful tests against the `Teststring`:

You can prefix the pattern string with a ‘!’ character (exclamation mark) to specify a non-matching pattern.

You can perform lexicographical string comparisons:

‘<CondPattern’ (lexicographically precedes)

Treats the CondPattern as a plain string and compares it lexicographically to TestString. True if TestString lexicographically precedes CondPattern.

```
# Only apply the rule if the requested host sorts before "m"
# (i.e. hostnames starting with a-l)
RewriteCond %{HTTP_HOST} <m
RewriteRule ^ /first-half-of-alphabet [L]
```

‘>CondPattern’ (lexicographically follows)

Treats the CondPattern as a plain string and compares it lexicographically to TestString. True if TestString lexicographically follows CondPattern.

```
# Redirect if the requested URI sorts after /wiki/
RewriteCond %{REQUEST_URI} >/wiki/
RewriteRule ^ /later-section [L]
```

‘=CondPattern’ (lexicographically equal)

Treats the CondPattern as a plain string and compares it lexicographically to TestString. True if TestString is lexicographically equal to CondPattern (the two strings are exactly equal, character for character). If CondPattern is "" (two quotation marks) this compares TestString to the empty string.

```
# Match only the exact hostname "www.example.com"
RewriteCond %{HTTP_HOST} =www.example.com
RewriteRule ^ /main-site/$0 [L]

# Check whether the query string is empty
RewriteCond %{QUERY_STRING} =""
RewriteRule ^/search$ /search?q=default [L]
```

‘<=CondPattern’ (lexicographically less than or equal to)

Treats the CondPattern as a plain string and compares it lexicographically to TestString. True if TestString lexicographically precedes CondPattern, or is equal to CondPattern (the two strings are equal, character for character).

```
# Match API versions up through "v3" (v1, v2, v3 but not v4)
RewriteCond %{HTTP:X-API-Version} <=v3
RewriteRule ^ /legacy-api%{REQUEST_URI} [L]
```

‘>=CondPattern’ (lexicographically greater than or equal to)

Treats the CondPattern as a plain string and compares it lexicographically to TestString. True if TestString lexicographically follows CondPattern, or is equal to CondPattern (the two strings are equal, character for character).

```
# Match API versions v3 and above
RewriteCond %{HTTP:X-API-Version} >=v3
RewriteRule ^ /modern-api%{REQUEST_URI} [L]
```

Note

These comparisons are *lexicographic* (byte-by-byte string ordering), not numeric. That means "9" > "10" is true, because "9" sorts after "1". If you need numeric comparisons, use the integer operators (-eq, -gt, etc.) described next.

Warning**Known bug in case-sensitive < and > operators**

The implementation of the case-sensitive < and > operators (the `compare_lexicography()` function in `mod_rewrite.c`) has a long-standing bug (Bug 40453): it compares string *lengths* first. When strings differ in length, the longer string is always considered “greater” regardless of content. This means "AAA" > "B" evaluates as **true**, which is incorrect.

Workarounds:

- Use the case-insensitive variants ([NC] flag or the <= / >= forms that use `strcasecmp`) — these are not affected.
- Use the `expr` syntax in a `RewriteCond`, which uses `ap_expr` and handles comparisons correctly:

```
RewriteCond expr "%{REQUEST_URI} < '/m'"
```

This bug has been present since at least `httpd 2.2` and remains unfixed as of `2.4.62`.

You can perform integer comparisons:

‘-eq’ (is numerically equal to)

The `TestString` is treated as an integer, and is numerically compared to the `CondPattern`. True if the two are numerically equal.

```
# Only apply to requests on port 8080
RewriteCond %{SERVER_PORT} -eq 8080
RewriteRule ^ /dev-portal%{REQUEST_URI} [L]
```

‘-ge’ (is numerically greater than or equal to)

The `TestString` is treated as an integer, and is numerically compared to the `CondPattern`. True if the `TestString` is numerically greater than or equal to the `CondPattern`.

```
# Redirect if the Content-Length is 10MB or more
RewriteCond %{HTTP:Content-Length} -ge 10485760
RewriteRule ^ /upload-too-large [R=413,L]
```

‘-gt’ (is numerically greater than)

The `TestString` is treated as an integer, and is numerically compared to the `CondPattern`. True if the `TestString` is numerically greater than the `CondPattern`.

```
# Route to the new server if the requested port is above 9000
RewriteCond %{SERVER_PORT} -gt 9000
RewriteRule ^ http://newserver.example.com%{REQUEST_URI} [R,L]
```

‘-le’ (is numerically less than or equal to)

The TestString is treated as an integer, and is numerically compared to the CondPattern. True if the TestString is numerically less than or equal to the CondPattern. Avoid confusion with the -l by using the -L or -h variant.

```
# Serve a lightweight page if the client says it can only accept
# small responses
RewriteCond %{HTTP:Max-Response-Size} -le 1024
RewriteRule ^/report$ /report-summary [L]
```

‘-lt’ (is numerically less than)

The TestString is treated as an integer, and is numerically compared to the CondPattern. True if the TestString is numerically less than the CondPattern. Avoid confusion with the -l by using the -L or -h variant.

```
# If the hour is before 06:00, show the overnight maintenance page
RewriteCond %{TIME_HOUR} -lt 06
RewriteRule ^ /overnight.html [L]
```

You can perform various file attribute tests:

‘-d’ (is directory)

Treats the TestString as a pathname and tests whether or not it exists, and is a directory.

```
# If the request maps to an existing directory, let it through
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]
```

‘-f’ (is regular file)

Treats the TestString as a pathname and tests whether or not it exists, and is a regular file.

```
# If the file doesn't exist, route to the front controller
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ /index.php [L]
```

‘-F’ (is existing file, via subrequest)

Checks whether or not TestString is a valid file, accessible via all the server’s currently-configured access controls for that path. This uses an internal subrequest to do the check, so use it with care - it can impact your server’s performance!

```
# Only rewrite if the target is actually accessible
RewriteCond /var/www/html%{REQUEST_URI} -F
RewriteRule ^/mirror/(.*)$ /$1 [L]
```

‘-H’ (is symbolic link, bash convention)

See -l.

‘-l’ (is symbolic link)

Treats the TestString as a pathname and tests whether or not it exists, and is a symbolic link. May also use the bash convention of -L or -h if there’s a possibility of confusion such as when using the -lt or -le tests.

```
# If the request points to a symlink, redirect to the real path
RewriteCond %{REQUEST_FILENAME} -l
RewriteRule ^(.*)$ /real$1 [R,L]
```

‘-L’ (is symbolic link, bash convention)

See -l.

‘-s’ (is regular file, with size)

Treats the TestString as a pathname and tests whether or not it exists, and is a regular file with size greater than zero.

```
# Serve cached content only if the cache file is non-empty
RewriteCond /var/cache/html%{REQUEST_URI} -s
RewriteRule ^(.*)$ /var/cache/html$1 [L]
```

‘-U’ (is existing URL, via subrequest)

Checks whether or not TestString is a valid URL, accessible via all the server’s currently-configured access controls for that path. This uses an internal subrequest to do the check, so use it with care - it can impact your server’s performance!

```
# Fall back to a mirror if the local URL would 404
RewriteCond %{REQUEST_URI} !-U
RewriteRule ^(.*)$ http://mirror.example.com$1 [R,L]
```

‘-x’ (has executable permissions)

Treats the TestString as a pathname and tests whether or not it exists, and has executable permissions. These permissions are determined according to the underlying OS.

```
# If the requested file is executable, run it as CGI
RewriteCond %{REQUEST_FILENAME} -x
RewriteRule ^/scripts/(.*)$ /cgi-bin/$1 [L]
```

Note:

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

If the TestString has the special value expr, the CondPattern will be treated as an ap_expr.

In the below example, -strmatch is used to compare the REFERER against the site hostname, to block unwanted hotlinking.

```
RewriteCond expr "! %{HTTP_REFERER} -strmatch '*://%{HTTP_HOST}/*'"
RewriteRule ^/images - [F]
```

Flag

You can also set special flags for CondPattern by appending [flags] as the third argument to the RewriteCond directive, where flags is a comma-separated list of any of the following flags:

‘nocase|NC’ (no case)

This makes the test case-insensitive - differences between ‘A-Z’ and ‘a-z’ are ignored, 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 condition/rule pair three times.

‘novary|NV’ (no vary)

If an HTTP header is used in the condition, this flag prevents this header from being added to the Vary header of the response.

Using this flag might break proper caching of the response if the representation of this response varies on the value of this header. So this flag should be only used if the meaning of the Vary header is well understood.

8.3 Examples

The following examples show RewriteCond in common real-world scenarios. Many of these appear again in *Recipes* with additional discussion.

8.3.1 Matching query strings

RewriteRule only matches against the URL-path — it never sees the query string. To test query string content, use RewriteCond with %{QUERY_STRING}:

```
# Redirect old query-string-based URLs to clean paths
RewriteCond %{QUERY_STRING} ^id=([0-9]+)$
RewriteRule ^/product$ /product/%1? [R=301,L]
```

This turns /product?id=42 into /product/42. The trailing ? in the substitution strips the original query string (without it, the query string is passed through by default). The %1 backreference comes from the RewriteCond capture group, not from RewriteRule.

8.3.2 Hostname-based routing

Test the Host: header to apply rules only to specific hostnames:

```
# Redirect www to non-www
RewriteCond %{HTTP_HOST} ^www\.example\.com$ [NC]
RewriteRule ^(.*)$ https://example.com$1 [R=301,L]
```

The [NC] flag on the condition makes the hostname comparison case-insensitive.

8.3.3 File and directory existence

The -f and -d tests check whether a path exists on disk. This is the basis of the front-controller pattern:

```
# If the request isn't an existing file or directory, route to index.php
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ /index.php [L]
```

The ! negates the test. Both conditions must be true (the default is AND), so the rule fires only when the request matches *neither* an existing file nor an existing directory.

Note: `FallbackResource` (see [FallbackResource](#)) does the same thing in a single line. Use `RewriteCond !-f` only when you need additional conditions or URL transformations that `FallbackResource` can't express.

8.3.4 Time-based rules

The `TIME_*` variables let you vary behavior by time of day, day of week, or date:

```
# Maintenance window: redirect all traffic between 2 AM and 4 AM
RewriteCond %{TIME_HOUR} ^0[2-3]$
RewriteRule !^/maintenance\.html$ /maintenance.html [R=302,L]
```

The `RewriteRule` pattern uses ! to exclude the maintenance page itself — without this, you'd create an infinite redirect loop.

8.3.5 Requiring HTTPS

Test whether the connection is secure:

```
RewriteCond %{HTTPS} !=on
RewriteRule ^(.*)$ https://%{HTTP_HOST}$1 [R=301,L]
```

Or, if you're behind a load balancer that terminates TLS and forwards `X-Forwarded-Proto`:

```
RewriteCond %{HTTP:X-Forwarded-Proto} !https
RewriteRule ^(.*)$ https://%{HTTP_HOST}$1 [R=301,L]
```

Note the syntax `%{HTTP:HeaderName}` for testing arbitrary HTTP request headers.

8.3.6 Combining conditions with OR

By default, multiple RewriteCond directives are ANDed. Use the [OR] flag for OR logic:

```
# Block two specific user agents
RewriteCond %{HTTP_USER_AGENT} BadBot [NC,OR]
RewriteCond %{HTTP_USER_AGENT} EvilScrapper [NC]
RewriteRule ^ - [F]
```

The [F] flag returns a 403 Forbidden. The rule fires if *either* condition matches.

REWRITEMAP

And memories, he knew, were not glass treasures to be
kept locked within a box. They were bright ribbons to
be hung in the wind.

—Terry Brooks, *The Talismans of Shannara*

The `RewriteMap` directive gives you a way to call external mapping routines to simplify a `RewriteRule`. This external mapping can be a flat text file containing one-to-one mappings, or a database, or a script that produces mapping rules, or a variety of other similar things. In this chapter we'll discuss how to use a `RewriteMap` in a `RewriteRule` or `RewriteCond`.

9.1 Creating a RewriteMap

The `RewriteMap` directive creates an alias which you can then invoke in either a `RewriteRule` or `RewriteCond` directive. You can think of it as defining a function that you can call later on.

The syntax of the `RewriteMap` directive is as follows:

```
RewriteMap MapName MapType:MapSource
```

Where the various parts of that syntax are defined as:

MapName

The name of the 'function' that you're creating

MapType

The type of the map. The various available map types are discussed below.

MapSource

The location from which the map definition will be obtained, such as a file, database query, or predefined function.

The `RewriteMap` directive must be used either in `virtualhost` context, or in `global server` context. This is because a `RewriteMap` is loaded at server startup time, rather than at request time, and, as such, cannot be specified in a `.htaccess` file.

9.2 Using a RewriteMap

Once you have defined a RewriteMap, you can then use it in a RewriteRule or RewriteCond as follows:

```
RewriteMap examplemap txt:/path/to/file/map.txt
RewriteRule ^/ex/(.*) ${examplemap:$1}
```

Note in this example that the RewriteMap, named ‘examplemap’, is passed an argument, \$1, which is captured by the RewriteRule pattern. It can also be passed an argument of another known variable. For example, if you wanted to invoke the examplemap map on the entire requested URI, you could use the variable %{REQUEST_URI} rather than \$1 in your invocation:

```
RewriteRule ^ ${examplemap:%{REQUEST_URI}}
```

9.3 Default Values

When a key is not found in the map, the lookup returns an empty string by default. You can specify a fallback using the pipe character (|) followed by a default value:

```
${mapname:key|default}
```

For example:

```
RewriteRule ^/product/(.*) /prods.php?id=${productmap:$1|NOTFOUND} [PT]
```

If the key \$1 is not found in productmap, the value NOTFOUND is substituted instead. This lets your application handle the missing-key case gracefully rather than receiving an empty string.

The default value can be any string, including a URL path:

```
RewriteRule ^/old/(.*) ${redirectmap:$1|/not-found.html} [R=301]
```

9.4 RewriteMap Types

There are a number of different map types which may be used in a RewriteMap.

9.4.1 int

An int map type is an internal function, pre-defined by mod_rewrite itself. There are four such functions:

9.4.2 toupper

The toupper internal function converts the provided argument text to all upper case characters.

```
# Convert any lower-case request to upper case and redirect
RewriteMap uc int:toupper
RewriteRule (.*?[a-z]+.*) ${uc:$1} [R=301]
```

9.4.3 tolower

The `tolower` is the opposite of `toupper`, converting any argument text to lower case characters.

```
# Convert any upper-case request to lower case and redirect
RewriteMap lc int:tolower
RewriteRule (.*[A-Z]+.*) ${lc:$1} [R=301]
```

9.4.4 escape

The `escape` internal function URL-encodes special characters in the argument, translating them to `%xx` hex sequences. This is useful when a captured backreference might contain characters that would break a URL — spaces, ampersands, question marks, and so on.

```
RewriteMap esc int:escape
RewriteRule ^/search/(.*)$ /search.php?term=${esc:$1} [PT]
```

A request for `/search/x & y` would result in the query string `term=x%20%26%20y`, which is properly encoded for use in a URL.

This is similar to what the `[B]` flag does to backreferences, but `escape` can be applied selectively to specific parts of the substitution via the map syntax, whereas `[B]` affects all backreferences in the rule.

9.4.5 unescape

The `unescape` internal function is the reverse of `escape` — it decodes `%xx` hex sequences back to their original characters.

```
RewriteMap unesc int:unescape
RewriteCond ${unesc:%{QUERY_STRING}} (*.secret.*)
RewriteRule ^ - [F]
```

This example decodes the query string before testing it, so that `%73ecret` is recognized as `secret` even when a client tries to sneak it past a filter using percent-encoding.

Use `unescape` when you need to inspect or match the decoded form of a URL component that may arrive in encoded form.

9.4.6 txt

A `txt` map is a plain text file containing one key-value pair per line, separated by whitespace. Lines starting with `#` are comments.

The file format looks like this:

```
##
## productmap.txt - Product name to ID mapping
##

television 993
```

(continues on next page)

(continued from previous page)

```
stereo      198
fishingrod 043
basketball  418
telephone   328
```

Define the map and use it in a rule:

```
RewriteMap productmap txt:/etc/httpd/maps/productmap.txt
RewriteRule ^/product/(.*) /prods.php?id=${productmap:$1|NOTFOUND} [PT]
```

A request for `/product/television` is internally rewritten to `/prods.php?id=993`. If the product isn't found in the map, the default value `NOTFOUND` is used instead (see *Default Values*).

Caching: httpd caches the contents of a `txt` map in memory. The cache is automatically refreshed when the file's modification time (`mtime`) changes, so you can update the file while the server is running — no restart required. However, for very large map files (thousands of entries), consider using a `dbm` map instead for faster lookups.

Context restriction: The `RewriteMap` directive itself must appear in server or virtual host context — you cannot declare it in a `.htaccess` file. However, once declared, the map can be *used* in `RewriteRule` and `RewriteCond` directives anywhere, including `.htaccess`.

9.4.7 rnd

A `rnd` map uses the same text file format as `txt`, but the value for each key is a pipe-separated list of alternatives. On each lookup, one value is chosen at random.

```
##
## servers.txt - Backend server map for load balancing
##

static www1.example.com|www2.example.com|www3.example.com|www4.example.com
dynamic app1.example.com|app2.example.com
```

Define the map and use it with the `[P]` flag for proxy load-balancing:

```
RewriteMap servers rnd:/etc/httpd/maps/servers.txt
RewriteRule ^/static/(.*) http://${servers:static}/$1 [P]
RewriteRule ^/app/(.*) http://${servers:dynamic}/app/$1 [P]
```

Each request for `/static/logo.png` is proxied to a randomly selected server from the `static` list.

Weighting trick: To weight the selection toward a particular server, list it more than once:

```
static www1|www1|www1|www2
```

This gives `www1` a 75% probability and `www2` a 25% probability.

Note that this is a very basic form of load balancing with no health checking or session affinity. For production load balancing, use `mod_proxy_balancer` instead. The `rnd` map is most useful for simple cases like

distributing static asset requests or A/B testing.

9.4.8 dbm

A `dbm` map stores the same key-value data as a `txt` map but in a DBM hash file, which provides $O(1)$ lookups instead of a linear scan. This matters when your map file has thousands of entries — a `txt` map is read sequentially, while a `dbm` lookup is essentially instantaneous regardless of file size.

Create a DBM file from a text file using the `httxt2dbm` utility that ships with `httpd`:

```
httxt2dbm -i redirectmap.txt -o redirectmap.map
```

Then reference it in your configuration:

```
RewriteMap redirects dbm:/etc/httpd/maps/redirectmap.map
RewriteRule ^/old/(.*) ${redirects:$1|/gone.html} [R=301]
```

You can optionally specify the DBM type:

```
RewriteMap redirects dbm=sdbm:/etc/httpd/maps/redirectmap.map
```

Available types include `sdbm` (always available), `gdbm`, `ndbm`, and `db`. In practice, use whatever `httxt2dbm` produces by default — it chooses the best available type for your platform.

Note: Some DBM implementations create two files (e.g., `redirectmap.map.dir` and `redirectmap.map.pag`). Always reference the base name without extensions in the `RewriteMap` directive.

Caching: Like `txt` maps, `dbm` maps are cached in memory and automatically refreshed when the file's modification time changes. To update a `dbm` map, regenerate it with `httxt2dbm` and `httpd` will pick up the new version on the next lookup.

9.4.9 prg

A `prg` map launches an external program at server startup and communicates with it via standard input and output. For each lookup, the key is written to the program's `STDIN` (followed by a newline), and the program writes the result to `STDOUT` (also followed by a newline).

To indicate that a key has no match, the program should return the string `NULL` (case-insensitive).

```
RewriteMap dash2under prg:/usr/local/bin/dash2under.py
RewriteRule - ${dash2under:%{REQUEST_URI}}
```

Here is an example program that replaces dashes with underscores:

```
#!/usr/bin/env python3
import sys

def main():
    for line in sys.stdin:
        key = line.strip()
        result = key.replace('-', '_')
```

(continues on next page)

(continued from previous page)

```
print(result, flush=True)

if __name__ == '__main__':
    main()
```

⚠ Warning

External program maps come with serious caveats:

- **Flush your output.** The program must flush STDOUT after every response line. Buffered output will cause httpd to hang waiting for a reply. In Python, use `flush=True` on `print()` or set `PYTHONUNBUFFERED=1`.
- **Single-process bottleneck.** Only one instance of the program runs. All requests that trigger the map are serialized through it. If the program is slow, it becomes a bottleneck for the entire server.
- **Hangs are fatal.** If the program blocks without responding, httpd will hang waiting for it. There is no timeout.
- **Crashes are permanent.** If the program dies, all subsequent lookups fail. You must restart httpd to relaunch it.
- **Startup only.** The program is launched when httpd starts (or restarts). It is not re-launched on failure.

You can run the program as a specific user and group by adding a third argument:

```
RewriteMap mymap prg:/path/to/program user:group
```

For most use cases, a `txt` or `dbm` map — or even `dbd` — is a better choice. Use `prg` only when you need logic that cannot be expressed as a static lookup table.

9.4.10 dbd

A `dbd` map looks up keys via a SQL query, using a database connection managed by `mod_dbd`. This lets you drive rewrite rules from a database table that can be updated in real time without touching configuration files or restarting the server.

There are two variants:

dbd

Executes the query on every lookup. Always returns the freshest data but incurs a database round-trip per request.

fastdbd

Caches results in memory after the first lookup. Faster, but cached entries are not refreshed until httpd is restarted. Use this when the data changes infrequently.

The `MapSource` is a SQL `SELECT` statement with `%s` as a placeholder for the lookup key:

```
RewriteMap myquery "fastdbd:SELECT destination FROM redirects WHERE source = %s"  
RewriteRule ^/r/(.*) ${myquery:$1|/not-found.html} [R=301]
```

You must also configure `mod_dbd` with a database connection:

```
DBDriver pgsq1  
DBDParams "host=dbhost dbname=mydb user=httpd password=secret"  
DBDMin 4  
DBDKeep 8  
DBDMax 20  
DBDExptime 300
```

If the query returns multiple rows, one is selected at random — similar to how `rnd` maps work. If no rows are returned, the default value (after `|`) is used.

When to use `dbd` vs `fastdbd`: Use `dbd` when the underlying data changes frequently and freshness matters (e.g., a vanity URL shortener updated by a CMS). Use `fastdbd` when the data is relatively stable and you want to minimize database load (e.g., a redirect table that's updated weekly).

`mod_dbd` is required — the `dbd` and `fastdbd` map types will produce a configuration error if it is not loaded.

PROXIES AND MOD_REWRITE

“His voice,” thought Will, “I never noticed. It’s the same color as his hair.”

—Ray Bradbury, *Something Wicked This Way Comes*

This chapter explores the intersection of `mod_rewrite` and `mod_proxy` — using rewrite rules to make proxying decisions dynamically.

Chapter 2 introduced `ProxyPass` and `ProxyPassReverse` as static URL mapping tools. Here we go further: using the `[P]` flag to proxy selectively, conditionally, and with URL transformations that `ProxyPass` alone cannot express.

10.1 The `mod_proxy` family

`mod_proxy` is the core module — it provides the framework and the `ProxyPass`, `ProxyPassReverse`, and `ProxyPassMatch` directives. By itself it does nothing; you pair it with one or more protocol modules that handle the actual communication with backends. These are the protocol modules available as of `httpd 2.5` (trunk):

HTTP/HTTPS backends:

`mod_proxy_http`

The workhorse. Proxies HTTP and HTTPS requests to backend servers. This is what you load for a standard reverse proxy.

`mod_proxy_http2`

Proxies requests to backends using HTTP/2. The backend must support HTTP/2 — there is no fallback to HTTP/1.1. Use this when your backend is an HTTP/2 server (e.g., gRPC services). 2.4.19 Available since `httpd 2.4.19`.

Application server protocols:

`mod_proxy_fcgi`

FastCGI support. The most common way to connect `httpd` to PHP-FPM, Python WSGI servers behind FastCGI adapters, and similar application servers. Supports both TCP and Unix domain sockets.

`mod_proxy_ajp`

Apache JServ Protocol (AJP/1.3) — primarily used with Apache Tomcat and other Java servlet containers. AJP is a binary protocol that’s more efficient than HTTP for proxy-to-backend communication, though it requires the backend to support it.

mod_proxy_uwsgi

The uWSGI protocol — used by the uWSGI application server, popular in the Python/Django ecosystem. 2.4.30 Available since httpd 2.4.30.

mod_proxy_scgi

The Simple Common Gateway Interface — a simpler alternative to FastCGI, used by some Python and Ruby application servers.

WebSockets and tunneling:

mod_proxy_wstunnel

WebSocket tunneling. Proxies WebSocket connections (`ws://` and `wss://`) to a backend WebSocket server. 2.4.5 Available since httpd 2.4.5. Note: since httpd 2.4.47, protocol upgrades (including WebSocket) can also be handled by `mod_proxy_http` directly — making `mod_proxy_wstunnel` less critical than it once was.

mod_proxy_connect

Handles the HTTP CONNECT method, used primarily for SSL/TLS tunneling through a forward proxy. If you're running a forward proxy that needs to pass HTTPS traffic, you need this.

mod_proxy_fdpass

Passes the client socket's file descriptor to another process via a Unix domain socket. A niche module for specialized architectures where another daemon handles the actual request. Unix only.

Legacy protocols:

mod_proxy_ftp

Proxies FTP requests. Allows clients to access FTP resources through the web server using HTTP. Limited to GET — you can retrieve files but not upload. Mostly a legacy feature at this point.

Load balancing and health checks:

mod_proxy_balancer

Adds load balancing across multiple backend servers. Supports several scheduling algorithms (by request count, by traffic, by busyness) via the `mod_lbmethod_*` modules. Includes a built-in web-based Balancer Manager for runtime configuration.

mod_proxy_hcheck

Dynamic health checking of backend workers. Periodically probes backends and automatically marks them as unavailable when they fail. Replaces the need for external health-check daemons in many configurations. 2.4.21 Available since httpd 2.4.21.

Dynamic configuration:

mod_proxy_express

Mass reverse proxying via DBM file lookup — maps hostnames to backends without per-host configuration. Covered in *mod_proxy_express*.

Response rewriting:

mod_proxy_html

Parses HTML responses and rewrites URLs in links, forms, and scripts so they point to the proxy rather than the backend. Covered in *Content Munging*.

10.1.1 Which modules to load

You always need `mod_proxy` itself plus at least one protocol module. A typical reverse proxy configuration loads:

```
LoadModule proxy_module      modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

For a PHP-FPM setup:

```
LoadModule proxy_module      modules/mod_proxy.so
LoadModule proxy_fcgi_module modules/mod_proxy_fcgi.so
```

For load-balanced backends with health checks:

```
LoadModule proxy_module      modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_hcheck_module modules/mod_proxy_hcheck.so
LoadModule lbmethod_byrequests_module modules/mod_lbmethod_byrequests.so
```

A common `mod_rewrite` pitfall: using the `[P]` flag without loading the appropriate protocol module. The rewrite will silently fail or produce a 500 error — check your error log for “No protocol handler was valid.”

10.2 When to use `[P]` vs ProxyPass

The `httpd` documentation itself says: “Consider using `ProxyPass` or `ProxyPassMatch` whenever possible in preference to `mod_rewrite`.” That’s good advice, and here’s why.

`ProxyPass` is a static mapping. It’s fast, simple, and handles connection pooling to the backend natively. The proxy worker is pre-configured at startup, so `httpd` can reuse connections, enforce timeouts, and manage retry logic without any per-request overhead:

```
ProxyPass      "/app" "http://backend:8080/app"
ProxyPassReverse "/app" "http://backend:8080/app"
```

The `[P]` flag on a `RewriteRule` achieves the same result, but it creates an *ad hoc* proxy request at the end of the rewrite processing — it doesn’t benefit from a pre-configured worker’s connection pool, and it’s evaluated later in the request lifecycle.

So when do you actually need `[P]`? When `ProxyPass` can’t express what you need:

- **Regex-based URL transformation** — you need to capture parts of the URL and rearrange them in the backend path.
- **Conditional proxying** — you want to proxy only when certain `RewriteCond` tests pass (a header value, a cookie, a missing local file).
- **Dynamic backend selection** — you’re using a `RewriteMap` to look up which backend to send the request to.

ProxyPassMatch covers some of the regex cases, but it can't do conditional logic. If you need both regex and conditions, [P] is your tool.

10.3 Basic proxying with [P]

The simplest [P] example looks like this:

```
RewriteEngine On
RewriteRule ^/app/(.*)$ http://backend:8080/$1 [P]
ProxyPassReverse "/app/" "http://backend:8080/"
```

A request for /app/dashboard is rewritten to http://backend:8080/dashboard and proxied to the backend.

Note the ProxyPassReverse — it's still required even when you're using [P] instead of ProxyPass. Here's why: when the backend sends a redirect response (a Location header like Location: http://backend:8080/login), ProxyPassReverse rewrites that header so the client sees /app/login instead of the internal backend URL. Without it, redirects from the backend will expose the backend's address to the client — or simply break, because the client can't reach the internal hostname.

10.4 Conditional proxying

This is where [P] really earns its keep — making the proxy decision based on conditions that ProxyPass can't evaluate.

Proxy when a local file doesn't exist — a common migration pattern where you're gradually moving content from a backend to the local server:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^/(.*)$ http://old-backend.internal/$1 [P]
ProxyPassReverse "/" "http://old-backend.internal/"
```

Requests for files that exist locally are served directly. Everything else is proxied to the old backend. As you migrate content, you simply add the files locally and they take precedence automatically.

Proxy based on a header or cookie — for example, routing beta users to a different backend:

```
RewriteEngine On
RewriteCond %{HTTP_COOKIE} beta=true
RewriteRule ^/(.*)$ http://beta-backend:8080/$1 [P]
ProxyPassReverse "/" "http://beta-backend:8080/"
```

Pattern-based backend selection — sending different URL paths to different backends:

```
RewriteEngine On
RewriteRule ^/api/(.*)$ http://api-server:8080/$1 [P]
RewriteRule ^/static/(.*)$ http://asset-server:9090/$1 [P]
```

This works, but be cautious about using it as a load balancer. If you're splitting traffic across multiple backends for the *same* path to distribute load, use `mod_proxy_balancer` instead — it handles health checks, failover, session stickiness, and scheduling algorithms that [P] rules can't replicate.

10.5 Proxying with RewriteMap

For truly dynamic backend selection, combine [P] with a RewriteMap (see [RewriteMap](#)). A text map file can map request paths — or any other variable — to backend URLs:

```
# /etc/httpd/conf/backend.map
widgets http://widget-server:8080
gadgets http://gadget-server:8080
default http://fallback-server:8080
```

```
RewriteMap backends txt:/etc/httpd/conf/backend.map
RewriteRule ^/store/([^/]+)/(.*)$ ${backends:$1|http://fallback-server:8080}/$2 [P]
```

A request for `/store/widgets/product/42` looks up `widgets` in the map and proxies to `http://widget-server:8080/product/42`.

For the simpler case of mapping hostnames to backends (mass virtual hosting), `mod_proxy_express` does this natively with a DBM file and no rewrite rules — see [mod_proxy_express](#). Reach for RewriteMap + [P] when you need more complex lookup logic than a straight hostname-to-backend mapping.

10.6 SSL/TLS considerations

When your backend uses HTTPS, you need to enable the SSL proxy engine:

```
SSLProxyEngine On
RewriteRule ^/secure/(.*)$ https://backend.internal/$1 [P]
```

By default, `httpd` will verify the backend's SSL certificate. If the backend uses a self-signed or internal CA certificate, you'll need to either provide the CA certificate or (in development only) disable verification:

```
# Point to your internal CA
SSLProxyCACertificateFile /etc/pki/tls/certs/internal-ca.pem

# OR, for development only - never in production
SSLProxyVerify none
SSLProxyCheckPeerCN off
SSLProxyCheckPeerName off
```

The redirect loop trap: a common pitfall when proxying between HTTP and HTTPS. If the backend redirects HTTP to HTTPS (as many applications do), and your proxy is also doing protocol translation, you can end up in an infinite redirect loop. The fix is usually to ensure the backend knows it's behind a proxy — either via `ProxyPreserveHost On` (so the backend sees the original `Host` header) or by setting the `X-Forwarded-Proto` header so the backend knows the client's original protocol:

```
RequestHeader set X-Forwarded-Proto "https"  
ProxyPreserveHost On
```

Most modern web frameworks check `X-Forwarded-Proto` and suppress their HTTP-to-HTTPS redirect when they see the client is already on HTTPS.

10.7 Rewriting proxied responses

`ProxyPassReverse` rewrites `Location` and other response *headers*, but it doesn't touch the response *body*. If your backend embeds hardcoded URLs in its HTML, you'll need `mod_proxy_html` or `mod_substitute` to fix them. These are covered in detail in *Content Munging*.

10.8 Common pitfalls

A quick rundown of the mistakes you'll make at least once (I certainly have):

Forgetting `ProxyPassReverse`. Everything appears to work until the backend sends a redirect, and the client gets a `Location` header pointing to your internal backend hostname. Always pair `[P]` with a matching `ProxyPassReverse`.

Not loading the right modules. `[P]` requires `mod_proxy` and the appropriate protocol module (`mod_proxy_http`, `mod_proxy_fcgi`, etc.). A missing module produces a 500 error and the log message "No protocol handler was valid for the URL."

Combining `[P]` and `[R]`. These flags are mutually exclusive. `[P]` proxies the request to the backend silently; `[R]` sends a redirect to the client. You can't do both. If you want to redirect and then have the *client's* new request be proxied, that's two separate rules.

DNS resolution timing. `ProxyPass` resolves the backend hostname at server startup. `RewriteRule ... [P]` resolves it at request time. This means `[P]` is more resilient to DNS changes (the backend IP can change without a server restart), but it also means a DNS lookup on every request — which can be slow if your DNS is unreliable. For `ProxyPass`, a `disablereuse=On` option forces per-request DNS, but at the cost of connection pooling.

Timeout and retry behavior. When a backend is slow or down, the default proxy timeout is inherited from the server's global `Timeout` directive (usually 60 seconds). You can tune this per-backend with `ProxyPass` parameters:

```
ProxyPass "/app" "http://backend:8080/app" timeout=10 retry=30
```

`timeout` controls how long to wait for the backend to respond; `retry` controls how long a failed backend is taken out of rotation before `httpd` tries it again. With `[P]` rules, these per-worker tuning options aren't available — another reason to prefer `ProxyPass` when you can.

VIRTUAL HOSTS AND MOD_REWRITE

When you've only got two ducks, they're always in a row.

—Rich Bowen

This chapter covers using `mod_rewrite` for dynamic virtual host configuration — mapping incoming hostnames to document roots, CGI directories, or entirely different server configurations on the fly.

Chapter 2 introduced `mod_vhost_alias` as the preferred tool for mass virtual hosting. This chapter shows the `mod_rewrite` approach, which offers more flexibility at the cost of more complexity. The `httpd` documentation itself advises: `mod_rewrite` is usually not the best way to configure virtual hosts — consider the alternatives first.

11.1 The problem

You have dozens — or hundreds, or thousands — of hostnames all pointing to the same server. Each hostname needs to serve content from a different directory. The traditional approach is a `<VirtualHost>` block for each one:

```
<VirtualHost *:80>
  ServerName www.alice.example.com
  DocumentRoot /var/www/vhosts/alice
</VirtualHost>

<VirtualHost *:80>
  ServerName www.bob.example.com
  DocumentRoot /var/www/vhosts/bob
</VirtualHost>

# ... repeat 500 more times
```

This doesn't scale. Every new hostname requires a config change and a server restart. For a shared hosting provider or a platform that provisions sites dynamically, you need the mapping to happen automatically — derive the document root from the hostname at request time, without any per-host configuration.

11.2 Dynamic vhosts with mod_rewrite

The core recipe uses `mod_rewrite` to capture the incoming hostname and map it to a filesystem path. Here's the full example:

```
RewriteEngine On

# Normalize the hostname to lowercase
RewriteMap lowercase int:tolower

# Capture the hostname and map it to a directory
RewriteCond %{HTTP_HOST} ^(.+)$
RewriteRule ^(.*)$ /var/www/vhosts/${lowercase:%1}/$1 [L]
```

Let's walk through this:

1. The `RewriteMap` defines a mapping called `lowercase` using the built-in `int:tolower` function — this normalizes the hostname so that `WWW.Example.COM` and `www.example.com` resolve to the same directory.
2. The `RewriteCond` captures the entire `Host` header into `%1`.
3. The `RewriteRule` captures the request path into `$1` and constructs the full filesystem path: `/var/www/vhosts/<hostname>/<path>`.

So a request for `http://www.alice.example.com/index.html` is served from `/var/www/vhosts/www.alice.example.com/index.html`.

Note

Remember the backreference distinction: `%1` through `%9` are captures from the most recent `RewriteCond`; `$1` through `$9` are captures from the `RewriteRule` pattern. Getting these mixed up is a very common mistake.

11.2.1 Stripping the `www.` prefix

If your directory structure doesn't include the `www.` prefix, strip it:

```
RewriteCond %{HTTP_HOST} ^(?:www\.)?(.+)$
RewriteRule ^(.*)$ /var/www/vhosts/${lowercase:%1}/$1 [L]
```

The non-capturing group `(?:www\.)?` matches and discards the `www.` if present, so `%1` contains only the bare domain.

11.3 Using a map file for vhosts

Deriving the path from the hostname with regex is convenient, but it assumes a predictable directory structure. When you need explicit control — mapping `customer-a.example.com` to `/var/www/sites/customer_a` rather than the hostname itself — a `RewriteMap` is cleaner:

```
# /etc/httpd/conf/vhost.map
customer-a.example.com /var/www/sites/customer_a
customer-b.example.com /var/www/sites/customer_b
demo.example.com      /var/www/sites/demo
```

```
RewriteMap vhostmap txt:/etc/httpd/conf/vhost.map
RewriteMap lowercase int:tolower

RewriteCond %{HTTP_HOST} ^(.+)$
RewriteCond ${vhostmap:${lowercase:%1}|NOTFOUND} !=NOTFOUND
RewriteRule ^(.*)$ ${vhostmap:${lowercase:%1}}/$1 [L]
```

The second `RewriteCond` checks that the hostname actually exists in the map — if it doesn't, the default `NOTFOUND` is returned and the rule is skipped, which lets the request fall through to a default virtual host.

For high-traffic servers, convert the text map to a DBM hash for faster lookups (see [RewriteMap](#) for the `httxt2dbm` tool and `dbm: map` type).

11.4 Handling aliases and CGI in dynamic vhosts

Here's a complication that catches everyone the first time: `mod_rewrite` runs before `mod_alias` in the request processing pipeline. That means `Alias` and `ScriptAlias` directives — things like `Alias /icons/ /usr/share/httpd/icons/` — get bypassed by the rewrite rule, because `mod_rewrite` has already mapped the URL to a filesystem path before `mod_alias` gets a chance.

The fix is to explicitly exclude those paths from rewriting:

```
# Let Alias and ScriptAlias handle these paths
RewriteCond %{REQUEST_URI} !^/icons/
RewriteCond %{REQUEST_URI} !^/cgi-bin/
RewriteCond %{REQUEST_URI} !^/error/

# Then the vhost mapping
RewriteCond %{HTTP_HOST} ^(.+)$
RewriteRule ^(.*)$ /var/www/vhosts/${lowercase:%1}/$1 [L]
```

Alternatively, if you need CGI to work within each virtual host's directory, you can use the `[H=cgi-script]` handler flag to force CGI processing for specific paths:

```
RewriteRule ^/cgi-bin/(.*)$ /var/www/vhosts/${lowercase:%1}/cgi-bin/$1 [H=cgi-
↪script,L]
```

This maps each vhost's `/cgi-bin/` to a directory within its own docroot and ensures the CGI handler runs.

11.5 Why mod_vhost_alias is usually better

Chapter 2 introduced `mod_vhost_alias`, and for mass virtual hosting, it's almost always the better choice. Here's why:

`mod_vhost_alias` maps hostnames to directories using interpolation tokens (`%0` for the full hostname, `%1/%2/etc.` for individual components), and it does so *before* the `Alias` and `ScriptAlias` resolution phase — meaning those directives still work correctly:

```
UseCanonicalName Off
VirtualDocumentRoot /var/www/vhosts/%0
```

That single directive replaces the entire `mod_rewrite` recipe above, including the lowercase map, the `RewriteCond`, and the exclusion conditions for `/icons/` and `/cgi-bin/`.

Use `mod_rewrite` for dynamic virtual hosts only when you need something `mod_vhost_alias` can't do:

- **Conditional logic** — different behavior for certain hostnames (e.g., redirect some to a different server, serve others from a special directory).
- **Hostname transformations** that go beyond token interpolation — for example, looking up the hostname in a database via a `RewriteMap` `prg`: external program.
- **Combining vhost mapping with other rewrite rules** — for example, forcing HTTPS *and* mapping to a vhost directory in the same ruleset.

11.6 Per-user virtual hosts

A common variant of dynamic hosting: mapping `~user` or `/users/username/` URLs to user home directories. Chapter 2 covered `mod_userdir`, which handles this natively:

```
UserDir public_html
```

This maps `~/alice/` to `/home/alice/public_html/`. Simple and effective.

The `mod_rewrite` version is useful when `mod_userdir` doesn't fit your layout — for example, if user content lives outside home directories, or you need conditional access:

```
# Map /users/alice/ to /var/www/users/alice/
RewriteRule ^/users/([^/]+)/?(.*)$ /var/www/users/$1/$2 [L]
```

Or with access restrictions:

```
# Only allow user pages for users in the map
RewriteMap validusers txt:/etc/httpd/conf/valid-users.txt
RewriteCond ${validusers:$1|INVALID} !=INVALID
RewriteRule ^/users/([^/]+)/?(.*)$ /var/www/users/$1/$2 [L]

# Everyone else gets a 404
RewriteRule ^/users/ - [R=404]
```

11.7 Logging for dynamic vhosts

When all your virtual hosts share a single configuration, they also share a single log file. The trick is to include the hostname in each log entry so you can distinguish them:

```
LogFormat "%V %h %l %u %t \"%r\" %>s %b" vhost_common
CustomLog /var/log/httpd/access_log vhost_common
```

The %V token logs the server name from the request (effectively the Host header). Now every log line is prefixed with the hostname.

Conditional logging — write different vhosts to different log files:

```
SetEnvIf Host "^alice\.example\.com$" vhost=alice
SetEnvIf Host "^bob\.example\.com$" vhost=bob

CustomLog /var/log/httpd/alice_access.log common env=vhost=alice
CustomLog /var/log/httpd/bob_access.log common env=vhost=bob
```

This works for a small number of known vhosts, but doesn't scale to the mass hosting scenario.

Splitting logs after the fact — for mass hosting, it's more practical to log everything to a single file with %V and split it later. Apache httpd ships with a `split-logfile` utility that reads a combined log and writes per-vhost log files:

```
split-logfile < /var/log/httpd/access_log
```

This creates files named `alice.example.com-access_log`, `bob.example.com-access_log`, and so on. It requires no per-vhost configuration, and handles the dynamic nature of mass hosting cleanly.

For real-time per-vhost logging at scale, piped logging through a script is also an option:

```
CustomLog "|/usr/local/bin/vhost-log-splitter.sh" vhost_common
```

But that's an exercise left to the reader — and to the reader's tolerance for debugging shell scripts in a production log pipeline.

ACCESS CONTROL WITH MOD_REWRITE

When the dragons grow too mighty
To slay with pen or sword
I grow weary of the battle
And the storm I walk toward
When all around is madness
And there's no safe port in view
I long to turn my path homeward
To stop awhile with you

—Rush, *Madrigal*

This chapter covers using `mod_rewrite` to control access to resources — blocking requests based on IP address, hostname, user agent, referrer, time of day, or other request characteristics.

As with other chapters, we'll show both the `mod_rewrite` approach and the simpler alternatives. In most access-control scenarios, `Require`, `SetEnvIf`, and `<If>` blocks are clearer and more maintainable than rewrite rules.

Note

The official Apache `httpd` documentation formerly had a dedicated `rewrite/access` guide for access-control recipes using `mod_rewrite`. That page has been deprecated and its content folded into the `rewrite/avoid` guide — a clear signal from the `httpd` project that these tasks are better handled by other means.

12.1 Forbidding image hotlinking

One of the oldest `mod_rewrite` recipes on the internet: blocking requests for your images when the `Referer` header indicates they're being embedded on someone else's site. The idea is that if another site links directly to your images, their visitors consume your bandwidth while you get none of the traffic.

Here's the basic approach — return a 403 Forbidden for any image request whose `Referer` doesn't match your own domain:

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^$
RewriteCond %{HTTP_REFERER} !^https?://(www\.)?example\.com [NC]
RewriteRule \.(png|jpg|gif|svg)$ - [F]
```

The first `RewriteCond` allows requests with an *empty* referer through — this covers direct visits, bookmarks, and privacy-conscious browsers that strip the header. The second condition blocks requests where the referer is *present* but doesn't match your domain. The `[NC]` flag makes the match case-insensitive.

Variante: serve an alternate image instead of a 403 — the classic “stop stealing my bandwidth” placeholder:

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^$
RewriteCond %{HTTP_REFERER} !^https?://(www\.)?example\.com [NC]
RewriteRule \.(png|jpg|gif|svg)$ /images/hotlink-denied.png [L]
```

Variante: redirect to the referring page's homepage:

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^$
RewriteCond %{HTTP_REFERER} !^https?://(www\.)?example\.com [NC]
RewriteRule \.(png|jpg|gif|svg)$ %{HTTP_REFERER} [R=302,L]
```

12.1.1 The non-rewrite alternative

You can do this without `mod_rewrite` using `SetEnvIf` and `Require`:

```
SetEnvIf Referer "^https?://(www\.)?example\.com" local_referer
SetEnvIf Referer "^$" local_referer

<FilesMatch "\.(png|jpg|gif|svg)$">
  Require env local_referer
</FilesMatch>
```

This is arguably clearer: the access decision is expressed as an authorization rule, not as a rewrite trick.

Limitations: The `Referer` header is optional — many clients don't send it, and some privacy tools strip it. It's also trivially spoofed by anyone determined to hotlink your images. This technique stops casual embedding but won't deter a motivated actor.

12.2 Blocking specific robots/user agents

To block a specific bot by user agent string, test `{HTTP_USER_AGENT}` and return a 403 with the `[F]` flag:

```
RewriteEngine On
RewriteCond %{HTTP_USER_AGENT} BadBot [NC]
RewriteRule ^ - [F]
```

To block multiple bots, chain them with `[OR]`:

```
RewriteEngine On
RewriteCond %{HTTP_USER_AGENT} BadBot [NC,OR]
RewriteCond %{HTTP_USER_AGENT} EvilScrapper [NC]
RewriteRule ^ - [F]
```

12.2.1 The non-rewrite alternative

```
SetEnvIfNoCase User-Agent "BadBot" bad_bot
SetEnvIfNoCase User-Agent "EvilScrapper" bad_bot
```

```
<RequireAll>
  Require all granted
  Require not env bad_bot
</RequireAll>
```

Why this is a weak defense: User agent strings are entirely client-controlled. Any bot that notices it's being blocked can simply change its user agent string to impersonate a legitimate browser. This technique is useful for blocking well-behaved bots that respect their own identity (and for blocking the occasional lazy scraper), but it's not a security measure.

For well-behaved bots, `/robots.txt` is the polite approach — it tells crawlers which paths to avoid without any server-side enforcement. For genuinely abusive traffic, escalate to firewall-level blocking (iptables, AWS WAF, fail2ban, or similar) where the decision is based on IP address and request patterns rather than a self-reported identity.

12.3 Denying by IP address or hostname

For a small number of addresses, inline `RewriteCond` tests work:

```
RewriteEngine On
RewriteCond %{REMOTE_ADDR} ^192\.168\.1\.100$
RewriteRule ^ - [F]
```

But for a larger deny list, a `RewriteMap` is more maintainable. Create a text file listing the blocked addresses:

```
# /etc/httpd/conf/denylist.txt
192.168.1.100 denied
10.0.0.50     denied
203.0.113.42 denied
```

Then use it in a `RewriteCond`:

```
RewriteMap denylist txt:/etc/httpd/conf/denylist.txt
RewriteCond ${denylist:%{REMOTE_ADDR}|OK} =denied
RewriteRule ^ - [F]
```

If the client's IP is found in the map, the lookup returns `denied` and the condition matches. If it's not found, the default value `OK` is returned and the request proceeds normally.

You can also check `%(REMOTE_HOST)` (the resolved hostname) in the same way, though this requires `HostnameLookups On`, which adds a DNS lookup to every request — a significant performance cost.

12.3.1 The modern alternative

Since `httpd 2.4`, the `Require` directive handles this natively and integrates with the authorization framework:

```
<RequireAll>
  Require all granted
  Require not ip 192.168.1.100 10.0.0.50 203.0.113.42
</RequireAll>
```

Or by hostname:

```
<RequireAll>
  Require all granted
  Require not host evil.example.com
</RequireAll>
```

This is shorter, clearer, and shows up where security reviewers expect to find access control rules. Use it.

12.4 Referrer-based deflection

Sometimes you want to redirect visitors who arrive from specific referring sites to a different URL — perhaps you’ve moved content and the old site still links to the wrong location, or you want to provide a landing page tailored to visitors from a particular source.

A `RewriteMap` keeps this clean when you have multiple referrers to handle:

```
# /etc/httpd/conf/referer-redirects.txt
old-partner.example.com    /welcome/partner
defunct-blog.example.net  /welcome/blog-readers
```

```
RewriteMap refmap txt:/etc/httpd/conf/referer-redirects.txt
RewriteCond %{HTTP_REFERER} ^https?://([^\s/]+)
RewriteCond ${refmap:%1|NONE} !=NONE
RewriteRule ^ ${refmap:%1} [R=302,L]
```

The first `RewriteCond` extracts the hostname from the referer into `%1`. The second looks it up in the map — if there’s no match, the default `NONE` is returned and the rule is skipped.

This is a legitimate technique for managing inbound traffic from specific sources. Less legitimate uses — redirecting competitors’ referral traffic, serving different content to visitors from review sites — tend to create a maintenance headache and erode trust. Use good judgment.

12.5 Time-based access control

mod_rewrite exposes time variables — `%{TIME_HOUR}`, `%{TIME_MIN}`, `%{TIME_WDAY}`, and others — that you can use in conditions. The most common use case is a maintenance window redirect:

```
# Redirect all traffic to a maintenance page between 2:00 and 4:00 AM
RewriteEngine On
RewriteCond %{TIME_HOUR} ^0[2-3]$
RewriteCond %{REQUEST_URI} !^/maintenance\.html$
RewriteRule ^ /maintenance.html [R=302,L]
```

The second condition prevents a redirect loop — without it, the request for `/maintenance.html` itself would also be redirected.

You can combine time conditions for more specific windows:

```
# Saturday (6) and Sunday (0) only
RewriteCond %{TIME_WDAY} ^[06]$
RewriteRule ^/office-hours - [F]
```

12.5.1 The `<If>` alternative

Since httpd 2.4, the `<If>` directive with `ap_expr` is a much cleaner way to express time-based logic:

```
<If "%{TIME_HOUR} -ge 2 && %{TIME_HOUR} -lt 4">
  RedirectMatch 302 !^/maintenance\.html$ /maintenance.html
</If>
```

This reads like what it is: a conditional block that applies during certain hours. No rewrite engine, no pattern matching — just a condition and an action.

12.6 Requiring HTTPS

This is the single most common mod_rewrite question on the mailing list — and the one where the alternatives are most compelling.

Approach 1: Redirect in a port-80 VirtualHost — the simplest and best option if you have access to the server config:

```
<VirtualHost *:80>
  ServerName www.example.com
  Redirect permanent "/" "https://www.example.com/"
</VirtualHost>
```

No rewrite engine, no conditions, no regex. Just a permanent redirect from HTTP to HTTPS. This is what you should use.

Approach 2: mod_rewrite — when you're in `.htaccess` and can't define virtual hosts:

```
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

Approach 3: <If> with ap_expr:

```
<If "! %{HTTPS} == 'on'">
  Redirect permanent "/" "https://www.example.com/"
</If>
```

All three accomplish the same thing. Approach 1 is preferred because it's the most explicit and doesn't involve pattern matching. Approach 2 is what you'll use in `.htaccess`. Approach 3 is a good middle ground when you have server config access but want it expressed as a condition.

See also the HTTP→HTTPS recipe in *Recipes*.

12.7 Environment variable gating

A powerful pattern in `httpd` configuration is to *separate the decision from the action*: one directive sets an environment variable based on some condition, and a later directive acts on that variable.

`SetEnvIf` and `BrowserMatch` are the typical variable-setters:

```
SetEnvIf Remote_Addr "^10\." internal_network
BrowserMatch "MSIE" legacy_browser
```

You can then test these variables in a `RewriteCond`:

```
RewriteEngine On
RewriteCond %{ENV:internal_network} ^$
RewriteRule ^/admin - [F]
```

This blocks access to `/admin` for anyone *not* on the internal network (where the variable would be empty). The logic lives in two places — the variable assignment and the rewrite condition — but it's compositional: you can reuse the same variable in multiple rules.

The `[E=varname:value]` flag on a `RewriteRule` works in the other direction — the rewrite rule sets a variable for downstream use:

```
RewriteRule ^/api/ - [E=api_request:1]
```

Other directives — `Header`, `CustomLog`, `<If>` — can then check `%{ENV:api_request}` to alter their behavior for API requests. This is a useful technique for making `mod_rewrite` cooperate with the rest of your configuration rather than trying to do everything itself.

12.8 When not to use mod_rewrite for access control

Throughout this chapter, every recipe has come with a non-rewrite alternative — and in most cases, the alternative is better. Here’s why.

mod_rewrite access control works, but it has real drawbacks:

- **It’s invisible to the authorization framework.** httpd has a well-defined authorization phase (Require, <RequireAll>, <RequireAny>). Rewrite rules run *before* that phase. Access restrictions expressed as rewrite rules don’t show up in the places where security reviewers expect to find them, and they don’t compose with other authorization rules.
- **It’s harder to audit.** A Require not ip 10.0.0.1 is self-documenting. A RewriteCond %{REMOTE_ADDR} followed by a RewriteRule ^ - [F] achieves the same thing in more lines with more room for subtle bugs.
- **It doesn’t integrate with logging.** When Require denies a request, the reason is logged through the authorization framework. When a RewriteRule returns [F], the log just shows a 403 with no explanation of *which* rule denied it — you’ll need to enable RewriteLog to trace it.

The httpd 2.4 alternatives cover nearly everything:

Require ip, Require not ip, Require host

IP and hostname-based access control, with full CIDR support.

<RequireAll>, <RequireAny>, <RequireNone>

Boolean composition of multiple access rules.

SetEnvIf + Require env

Header-based decisions (user agent, referer, custom headers).

<If> with ap_expr

Arbitrary conditions: time of day, SSL variables, request characteristics, even complex boolean expressions.

Use mod_rewrite for access control only when you genuinely need something the authorization framework can’t express — which, in httpd 2.4 and later, is rare.

CONDITIONAL CONFIGURATION

Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke, *Profiles of the Future*

13.1 Introduction

While the Apache httpd configuration files have always had some ways to make things conditional, with the advent of version 2.4, there's an explosion in the ways that you can make your configuration file reactive and programmable. That is, you can make your configuration more responsive to the specifics of the request that it servicing.

In this part of the book, we discuss some of this functionality. Some of it is specific to version 2.4 and later, while some of it has been available for years.

13.2 Match Directives

Several Apache httpd directives have `Match` variants that accept regular expressions instead of simple strings. If you're comfortable with regex (and by now you should be — you read Chapter 1), these give you a lot of power without reaching for `mod_rewrite`.

FilesMatch

Works like `<Files>`, but matches filenames against a regex:

```
# Deny access to editor backup files
<FilesMatch "\.(bak|orig|swp)$">
    Require all denied
</FilesMatch>
```

DirectoryMatch

Like `<Directory>`, but with a regex path:

```
<DirectoryMatch "^/var/www/[0-9]{4}/archive">
    Options -Indexes
</DirectoryMatch>
```

LocationMatch

Like `<Location>`, but with a regex URL-path:

```
<LocationMatch "^/api/v[0-9]+/">
  Header set X-API true
</LocationMatch>
```

RedirectMatch

Redirect with regex support and backreferences:

```
# /docs/2023/report.html → /archive/2023/report.html
RedirectMatch 301 "^/docs/([0-9]{4})/(.*)$" "/archive/$1/$2"
```

ScriptAliasMatch

Maps URL patterns to CGI directories:

```
ScriptAliasMatch "^/cgi-bin/(admin|user)/(.*)$" "/usr/local/cgi/$1/$2"
```

These are all worth knowing about because they're often a better fit than a `RewriteRule` for straightforward pattern-based matching. They make the configuration's intent clear to anyone reading it, and they don't require `RewriteEngine On`.

13.3 IfDefine

The `IfDefine` directive provides a way to make blocks of your configuration file optional, depending on the presence, or absence, of an appropriate command-line switch. Specifically, a configuration block wrapped in an `<IfDefine XYZ>` container will be invoked if and only if the server is started up with a `-D XYZ` command line switch.

Consider, for example a configuration as follows:

```
<IfDefine TEST>
  ServerName test.example.com
</IfDefine>
<IfDefine !TEST>
  ServerName www.example.com
</IfDefine>
```

Now, you can start the server with a `-D TEST` command line option:

```
httpd -D TEST -k restart
```

This will result in the first of the two `IfDefine` blocks being loaded. Conversely, if you omit the `-D TEST` flag, the server will start with the second of the two `IfDefine` blocks loaded.

This gives the ability to keep several configurations in the same file, and load various components on demand. You could even deploy the same configuration file to several different servers, but start each with different command line flags (you can specify more than one `-D` flag at startup) to start the servers up in different configurations.

`<IfDefine>` blocks can be nested, so that you can combine several conditions, as seen in this example from the docs:

```

<IfDefine ReverseProxy>
  LoadModule proxy_module    modules/mod_proxy.so
  LoadModule proxy_http_module modules/mod_proxy_http.so
  <IfDefine UseCache>
    LoadModule cache_module modules/mod_cache.so
    <IfDefine MemCache>
      LoadModule mem_cache_module modules/mod_mem_cache.so
    </IfDefine>
    <IfDefine !MemCache>
      LoadModule cache_disk_module modules/mod_cache_disk.so
    </IfDefine>
  </IfDefine>
</IfDefine>

```

You could then, for example, start the server up with:

```
httpd -DReverseProxy -DUseCache -DMemCache -k restart
```

(The space between `-D` and the flag is optional.)

13.4 Define

New with the 2.3 (and later) version of the server is the `Define` directive, which lets you define variables within the configuration file, which can then be used later on in the configuration, either as part of a configuration directive, or in an `<IfDefine ...>` directive.

Consider this variation on the earlier example:

```

<IfDefine TEST>
  Define servername test.example.com
</IfDefine>
<IfDefine !TEST>
  Define servername www.example.com
  Define SSL
</IfDefine>

DocumentRoot /var/www/${servername}/htdocs

```

A variable `VAR` defined with the `Define` directive can then be used later using the `${VAR}` syntax, as shown here. In the case where no value is given (see the line `Define SSL`) the variable is set to `TRUE`, which can then be tested later using an `<IfDefine>` test.

In this example, as before, the server should be started with a `-DTEST` command line option to use the first definition of `servername` and without it to use the second.

Or you can use a `Define` directive to define something, such as a file path, which is then used several times in the configuration:

```
Define docroot /var/www/vhosts/www.example.com

DocumentRoot ${docroot}

<Directory ${docroot}>
    Require all granted
</Directory>
```

13.5 <If>, <Elsif>, and <Else>

New in Apache httpd 2.4 is the ability to put <If> blocks in your configuration file to make it truly conditional. This provides a level of flexibility that was never before available.

Whereas the <IfDefine> and <Define> directives are evaluated at server startup time, <If> is evaluated at request time, giving you the chance to make configuration dependant on values that may change from one HTTP request to another. Naturally, this results in some request-time overhead, but the flexibility that you gain may be worth this to you in some situations.

Consider the following examples to give you some ideas:

13.5.1 Canonical hostname

In many situations, it is desirable to enforce a particular hostname on your website. For example, if you are setting cookies, you need to ensure that those cookies are valid for all requests to your site, which requires that the hostname being accessed match the hostname on the cookie itself. So, when someone accesses your site using the hostname `example.com`, you want to redirect that request to use the hostname `www.example.com`.

In previous versions of httpd, you may have used `mod_rewrite` to perform this redirection, but <If> provides a more intuitive syntax:

```
# Compare the host name to example.com and
# redirect to www.example.com if it matches
<If "%{HTTP_HOST} == 'example.com'">
    Redirect permanent / http://www.example.com/
</If>
```

13.5.2 Image hotlinking

You may wish to prevent another website from embedding your images in their pages - so-called image hotlinking. This is usually done by comparing the `HTTP_REFERER` variable on a request to these images to ensure that the request originated within a page on your site:

```
# Images ...
<FilesMatch "\.(gif|jpe?g|png)$">
    # Check to see that the referer is right
    <If "%{HTTP_REFERER} !~ /example.com/" >
        Require all denied
```

(continues on next page)

(continued from previous page)

```
</If>
</FilesMatch>
```

13.5.3 The expression parser (ap_expr)

The <If> examples above use a powerful expression syntax called `ap_expr`. This is httpd 2.4's general-purpose expression parser, and it appears in more places than just <If> blocks:

- <If>, <ElseIf> — conditional configuration sections
- RewriteCond with the expr TestString (see *RewriteCond*)
- SetEnvIfExpr — conditional environment variables
- CustomLog with expr= — conditional logging
- Header and RequestHeader with expr=
- <AuthzProviderAlias> and Require expr

The expression language supports:

- **String comparisons:** ==, !=, <, >, <=, >=
- **Regex matching:** =~, !~
- **Integer comparisons:** -eq, -ne, -lt, -le, -gt, -ge
- **Unary file tests:** -d, -f, -s, -L, -e, -F, -U (same ones available in RewriteCond)
- **IP matching:** -ipmatch for CIDR notation (e.g. -ipmatch '10.0.0.0/8')
- **String functions:** tolower(), toupper(), escape(), base64(), md5(), sha1(), file(), filesize()
- **Request functions:** req('Header-Name'), resp(), reqenv(), osenv(), note()
- **Boolean logic:** &&, ||, !, parentheses for grouping
- **List operator:** word -in {list}

I covered the comparison operators and file tests in detail in *RewriteCond*, since that's where most people first encounter them. The same operators work identically in `ap_expr` contexts.

The full expression syntax is documented at <https://httpd.apache.org/docs/2.4/expr.html>. It's a dense reference page, but it's comprehensive. Bookmark it — you'll refer to it often once you start using <If> and SetEnvIfExpr in earnest.

13.6 mod_macro

`mod_macro` has been around for a while, but with the 2.4 version of the server it is now one of the modules that comes with the server itself, rather than being a third-party module obtained and installed separately.

It provides the ability - as the name suggests - to create macros within your configuration file, which can then be invoked multiple times, in order to produce several similar configuration blocks. Parameters can be provided to fill in the variables in those macros.

Macros are evaluated at server startup time, and the resulting configuration is then loaded as though it was a static configuration file on disk.

13.7 mod_proxy_express

`mod_proxy_express` is a mass reverse-proxy engine. Where `mod_proxy` with `ProxyPass` requires you to define each backend explicitly, `mod_proxy_express` reads a DBM map file and routes requests based on the incoming `Host` header — no per-host configuration blocks needed.

The setup is minimal. First, create a plain text map:

```
# express-map.txt
www1.example.com http://192.168.211.2:8080
www2.example.com http://192.168.211.12:8088
www3.example.com http://192.168.212.10
```

Convert it to a DBM file:

```
httxt2dbm -i express-map.txt -o express-map
```

Then enable it in your server config:

```
ProxyExpressEnable on
ProxyExpressDBMFile /path/to/express-map
```

That's the entire configuration. A request with `Host: www2.example.com` is proxied to `http://192.168.211.12:8088`. Adding a new backend is a matter of adding a line to the text file, re-running `httxt2dbm`, and doing a graceful restart.

This module doesn't support regex or pattern matching — it's a straight hostname-to-backend lookup. If you need more sophisticated routing (path-based proxying, load balancing, failover), use `ProxyPass` and `ProxyPassReverse` directly. But for the "I have hundreds of domains that each map to a backend" case, `mod_proxy_express` is hard to beat.

13.8 mod_vhost_alias

`mod_vhost_alias` dynamically maps hostnames (or IP addresses) to document roots using interpolation patterns, without defining individual `<VirtualHost>` blocks. It's the dedicated, purpose-built solution for mass virtual hosting.

The core directive is `VirtualDocumentRoot`. You give it a pattern string, and it constructs the document root from parts of the requested hostname:

```
UseCanonicalName Off
VirtualDocumentRoot /var/www/vhosts/%0
```

A request for `http://blog.example.com/index.html` serves `/var/www/vhosts/blog.example.com/index.html`. The `%0` is replaced by the full hostname.

The interpolation syntax supports extracting parts of the hostname by position (`%1` for the first dot-separated component, `%2` for the second, and so on) and even individual characters within those parts. This lets you create directory hierarchies that spread the load across the filesystem:

```
# blog.example.com → /var/www/vhosts/example.com/b/l/blog
VirtualDocumentRoot /var/www/vhosts/%3+/%2.1/%2.2/%2
```

There's also `VirtualScriptAlias` for CGI directories, and IP variants of both directives (`VirtualDocumentRootIP`, `VirtualScriptAliasIP`) that interpolate based on the server's IP address rather than the hostname.

I covered this module briefly in *Virtual hosts and mod_rewrite* and recommended it over `mod_rewrite`-based mass vhosting in most cases. The reason it belongs in this chapter too is that it's a prime example of configurable configuration — one line replaces thousands of `<VirtualHost>` blocks.

13.9 Conditional logging

There are times when you want to exclude certain requests from your access logs — health checks from a load balancer, requests for `robots.txt`, asset requests that would drown out the interesting traffic. Apache's `CustomLog` directive supports this through environment variable conditions and, since 2.4, through expressions.

The basic idea: set an environment variable on requests you want to filter, then use the `env=` clause on `CustomLog` to include or exclude those requests. You can also use conditional format strings that log different fields depending on the HTTP response code.

I wrote the original version of this section in the `httpd` docs, and it remains one of the more useful but under-discovered features.

13.9.1 env=

The `env=` clause on `CustomLog` includes or excludes log entries based on whether an environment variable is set. There are four directives for setting these variables, each suited to different situations.

`SetEnv` (from `mod_env`) unconditionally sets a variable on every request. It's useful as a baseline that other modules may clear:

```
# Set on every request - mod_cache will suppress it on cache hits
SetEnv CACHE_MISS 1
```

`SetEnvIf` (from `mod_setenvif`) sets a variable conditionally, based on a request attribute and a regex. The attribute can be any HTTP request header (`User-Agent`, `Referer`, `Accept-Language`, etc.), or one of the special attributes `Remote_Addr`, `Remote_Host`, `Request_Method`, `Request_Protocol`, or `Request_URI`:

```
# Mark requests from loopback and for robots.txt
SetEnvIf Remote_Addr "127\.0\.0\.1" dontlog
```

(continues on next page)

(continued from previous page)

```
SetEnvIf Request_URI "^/robots\.txt$" dontlog

# Match a specific browser family
SetEnvIf User-Agent "Googlebot" is_bot
```

You can also set the value, unset a variable with `!`, or set multiple variables in a single directive:

```
SetEnvIf Request_URI "^/api/" api_request !dontlog source=api
```

`SetEnvIfNoCase` works identically to `SetEnvIf` but the regex match is case-insensitive — useful for header values where case varies:

```
SetEnvIfNoCase User-Agent "googlebot" is_bot
SetEnvIfNoCase User-Agent "bingbot" is_bot
```

`SetEnvIfExpr` (available since 2.4.2) uses an `ap_expr` expression instead of an attribute/regex pair, giving you access to the full expression syntax — comparisons, functions, boolean logic, IP matching, and more. See *The expression parser (ap_expr)* above for what's available in an expression:

```
# Mark slow requests (> 5 seconds)
SetEnvIfExpr "%{DURATION} -ge 5000000" slow_request

# Mark internal network requests
SetEnvIfExpr "-R '10.0.0.0/8'" internal
```

Once you've set variables, reference them in `CustomLog` with `env=` (include) or `env=!` (exclude):

```
# Log everything except loopback and robots
CustomLog logs/access_log common env=!dontlog

# Separate log for bots
CustomLog logs/bot_log common env=is_bot
```

The `!` negates the test — log the request only if `dontlog` is *not* set.

You can also split traffic into separate logs. For example, logging English-speaking visitors to one file and everyone else to another:

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

A clever use of this technique is measuring cache efficiency. Set an environment variable that `mod_cache` will suppress on a cache hit:

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

Because `mod_cache` runs before `mod_env`, a cache hit serves the content without `mod_env` ever executing — so `CACHE_MISS` logs as `-`. A cache miss lets `mod_env` run normally, logging `1`. Scan the log for the ratio of dashes to ones and you have your hit rate.

13.9.2 Response-code conditional format strings

`LogFormat` also supports logging values *conditional on the HTTP response code*. Prefix the format token's header name with a comma-separated list of status codes:

```
# Log User-Agent only for 400 and 501 responses
LogFormat "%400,501{User-agent}i" browserlog

# Log Referer only for non-success responses
LogFormat "%!200,304,302{Referer}i" refererlog
```

In the first example, the `User-agent` is logged if the response status is `400` or `501`; otherwise a literal `-` is logged. In the second, the `Referer` is logged if the status is *not* `200`, `304`, or `302` — note the `!` before the status codes.

This is useful for keeping logs compact. If you only care about the referring page when something goes wrong, there's no reason to log it on every successful request.

Since `httpd 2.4`, you can also use `expr=` for more complex conditions without a separate `SetEnvIf`:

```
# Log requests that took more than 5 seconds
CustomLog logs/slow_log combined "expr=%D -ge 5000000"
```

Although conditional logging is powerful, it's worth noting that log files are generally more useful when they contain a *complete* record of server activity. It's often easier to log everything and post-process the logs to remove what you don't need, rather than trying to predict at configuration time what you'll want to see later.

13.9.3 Per-module logging

As I discussed in *Rewrite Logging*, the `LogLevel` directive accepts per-module granularity. This isn't limited to `mod_rewrite` — you can set the log level for *any* module:

```
# Global level warn, but trace SSL handshakes and rewrite processing
LogLevel warn ssl:info rewrite:trace3
```

You can reference a module by its identifier (`ssl_module`), its source filename (`mod_ssl.c`), or the shorthand form without the `_module` suffix (`ssl`). All three are equivalent.

This is invaluable for debugging — you can turn up the verbosity of one module without drowning in output from every other module on the server.

13.9.4 Per-directory logging

Since `httpd 2.3.6`, `LogLevel` can also be set inside `<Directory>`, `<Location>`, and `<VirtualHost>` blocks. This lets you debug a specific path or `vhost` without flooding the global error log:

```
# Debug rewrites only for the /checkout/ path
<Location "/checkout/">
    LogLevel warn rewrite:trace4
</Location>
```

I covered this in detail in *Per-Directory Logging* in the Rewrite Logging chapter. The key thing to remember is that per-directory log level changes only affect messages generated *after* the request has been parsed and associated with that directory context. Connection-level messages are still controlled by the server-wide LogLevel.

13.9.5 Piped logging

Instead of writing to a file, you can pipe log output to an external program. The most common use is `rotatelogs`, which ships with `httpd` and handles automatic log rotation:

```
# Rotate the access log every 24 hours (86400 seconds)
CustomLog "|/usr/bin/rotatelogs /var/log/httpd/access_log.%Y%m%d 86400" combined

# Rotate when the log reaches 10MB
CustomLog "|/usr/bin/rotatelogs /var/log/httpd/access_log 10M" combined
```

The pipe character (`|`) tells `httpd` to spawn the program and send log lines to its standard input. You can pipe to any program — people use this for real-time log analysis, forwarding to `syslog`, or feeding into monitoring pipelines.

A few things to watch out for:

- If the piped program crashes, `httpd` may stop logging (or buffer and retry, depending on the implementation). Use `||` (two pipes) to tell `httpd` to restart the program if it dies.
- Piped logging has a small performance overhead compared to direct file writes. For most sites it's negligible, but on very high-traffic servers it's worth benchmarking.
- The `BufferedLogs` directive can batch log writes for better I/O performance, but be aware that a crash could lose buffered entries.

For production use, `rotatelogs` with the `-l` flag (use local time instead of UTC) and `-f` flag (force opening the log file on startup) covers most rotation needs without third-party tools like `logrotate`.

CONTENT MUNGING

On the Coast of Coromandel

Where the early pumpkins blow,
In the middle of the woods
Lived the Yonghy-Bonghy-Bo.

—Edward Lear, *The Courtship of the Yonghy-Bonghy-Bo*

Everything we’ve discussed so far in this book transforms the *request* — the URL, the query string, headers, environment variables. The server decides *what* to serve, but the content itself passes through untouched.

Sometimes that’s not enough. You may need to modify the *response* — the actual HTML, CSS, or other content that the server sends back to the client. Perhaps you’ve placed a reverse proxy in front of a backend application and the HTML it returns is full of hardcoded URLs pointing to the backend’s internal hostname. Or perhaps you need to inject a tracking snippet into every page, or strip out a legacy banner that the backend still inserts.

`mod_rewrite` can’t help you here. It has already done its work by the time the response body is being assembled. For response transformation, you need the output filter modules. We’re going to look at three of them — `mod_substitute`, `mod_sed`, and `mod_proxy_html` — and then at the filter framework that ties them all together.

14.1 `mod_substitute`

`mod_substitute` performs search-and-replace on the response body using a syntax borrowed from `sed`. It’s the simplest of the content transformation modules, and the one you’ll reach for most often.

The core directive is `Substitute`, which takes a substitution expression in the familiar `s/pattern/replacement/flags` form. Any single character can serve as the delimiter — you don’t have to use `/` — which is handy when your patterns contain slashes, as URLs tend to.

14.1.1 Basic usage

The most common use case is rewriting URLs in proxied content. Suppose you have a reverse proxy in front of a backend server at `http://backend.internal:8080`. The backend generates HTML with absolute URLs pointing to itself:

```
<a href="http://backend.internal:8080/dashboard">Dashboard</a>  

```

You need these rewritten so they point to your public-facing proxy. ProxyPassReverse handles Location and other *headers*, but it doesn't touch the *body*. That's where Substitute comes in:

```
ProxyPass      "/app" "http://backend.internal:8080"
ProxyPassReverse "/app" "http://backend.internal:8080"

<Location "/app">
  AddOutputFilterByType SUBSTITUTE text/html
  Substitute "s|http://backend.internal:8080|/app|ni"
</Location>
```

Let's break down the flags:

n

Treat the pattern as a literal string, not a regular expression. This is what you want when you're matching a fixed URL — no need to worry about escaping dots and slashes.

i

Case-insensitive match. Useful because some backends generate mixed case in URLs (HTTP:// vs http://).

Other available flags are:

f

Flatten line breaks. Normally, mod_substitute processes the response one line at a time. The f flag collapses the entire response (or the current buffer) into a single line before applying the substitution. Use this when the text you're trying to match spans multiple lines — for example, an HTML tag with attributes split across lines.

q

Quote (escape) special characters in the replacement string. Use this when your replacement contains characters that would otherwise be interpreted as regex backreferences.

You can stack multiple Substitute directives — they're applied in order:

```
<Location "/app">
  AddOutputFilterByType SUBSTITUTE text/html
  Substitute "s|http://backend.internal:8080|/app|ni"
  Substitute "s|backend.internal|www.example.com|ni"
</Location>
```

14.1.2 Handling long lines

By default, mod_substitute processes lines up to 1 MiB in length. If your backend produces minified HTML or JSON where the entire response is a single line, you may hit this limit. Increase it with:

```
SubstituteMaxLineLength 10m
```

The value accepts k (kilobytes) and m (megabytes) suffixes.

14.1.3 Using regular expressions

When literal matching isn't enough, drop the `n` flag and use a full regular expression. For example, to strip all `onclick` attributes from the response:

```
Substitute "s/\s+onclick=\"[^\"]*\"//i"
```

Or to rewrite a family of image URLs:

```
Substitute "s|/old-images/(.*?\.(?:png|jpg|gif))|/new-images/$1|i"
```

Be conservative with regex substitutions on response bodies. A poorly-anchored pattern applied to a large HTML document can produce surprising results — or silently corrupt content. Always test thoroughly, and prefer the `n` (literal) flag when exact string matching will do.

14.1.4 Content type filtering

Notice the `AddOutputFilterByType` directive in the examples above. This is important: you almost certainly don't want to run text substitutions on binary content like images or PDFs. Always scope your `Substitute` directives to the appropriate content types:

```
# Only apply to HTML and CSS
AddOutputFilterByType SUBSTITUTE text/html text/css
```

If you need to apply substitutions to *all* text content, you can add `text/plain`, `application/javascript`, `application/json`, and so on — but think twice before casting too wide a net.

14.2 mod_sed

Where `mod_substitute` gives you a single `s/find/replace/` operation per directive, `mod_sed` gives you the full `sed` stream-editing language. It's more powerful, but also more complex, and you'll need it far less often.

`mod_sed` provides two directives:

OutputSed

Applies a `sed` command to the response body (output filter).

InputSed

Applies a `sed` command to the request body (input filter). This is unusual — it means you can transform POST data before your application sees it. There aren't many use cases for this in practice, but it's there if you need it.

14.2.1 Basic usage

Here's a simple example: stripping all `<script>` blocks from proxied HTML, perhaps as a crude XSS mitigation on a legacy backend:

```
<Location "/legacy">
  AddOutputFilterByType Sed text/html
```

(continues on next page)

(continued from previous page)

```
OutputSed "s/<script[^>]*>.*<\/script>\/gi"  
</Location>
```

Note the filter name is Sed (capital S), not SUBSTITUTE.

14.2.2 When to reach for mod_sed

For simple search-and-replace, `mod_substitute` is easier and sufficient. Reach for `mod_sed` when you need capabilities that go beyond a single substitution:

- **Address ranges:** Apply a command only to lines matching a pattern or falling within a line range. For example, transform only lines inside a `<div class="legacy">` block.
- **Multiple commands:** Chain several `sed` commands in sequence. You can use `OutputSed` multiple times, or separate commands with semicolons if your `sed` syntax supports it.
- **Delete operations:** The `d` command deletes entire lines matching a pattern, which is cleaner than substituting them with an empty string.

```
# Delete all HTML comment lines from the response  
OutputSed "/^<!--/d"
```

In practice, if you find yourself writing complex `sed` programs inside your Apache configuration, it's worth stepping back and asking whether the transformation should happen at the application level instead. A few lines of `sed` in the config is fine; a 20-line `sed` script embedded in `OutputSed` directives is a maintenance hazard.

14.3 mod_proxy_html

`mod_proxy_html` takes a fundamentally different approach from the two modules above. Instead of doing blind text search-and-replace, it actually *parses* the HTML and rewrites only the URLs it finds in elements and attributes. It understands `<a href>`, ``, `<form action>`, `<link href>`, and all the other places where URLs appear in HTML.

This makes it both safer and more precise for URL rewriting — it won't accidentally mangle text that happens to look like a URL, because it only touches actual attribute values in HTML elements. On the other hand, it's useless for non-HTML content, and it does nothing for URLs embedded in JavaScript strings or inline CSS (unless you enable `ProxyHTMLExtended`).

14.3.1 A worked example

Here's the classic reverse proxy scenario. You proxy `/app` to a backend at `http://backend:8080/`, and the backend generates HTML with absolute paths rooted at `/`:

```
<a href="/dashboard">Dashboard</a>  
<link rel="stylesheet" href="/css/style.css">  
<form action="/api/submit" method="post">
```

These paths are wrong from the client's perspective — the client needs to see `/app/dashboard`, `/app/css/style.css`, and `/app/api/submit`. Here's how `mod_proxy_html` handles it:

```
ProxyPass          "/app" "http://backend:8080"
ProxyPassReverse  "/app" "http://backend:8080"

<Location "/app">
  ProxyHTMLEnable On
  ProxyHTMLURLMap "/" "/app/"
</Location>
```

`ProxyHTMLURLMap` takes a *from* pattern and a *to* string, similar to `Substitute` but operating on parsed URL attributes rather than raw text. This single directive will rewrite every `href`, `src`, `action`, and similar attribute in the HTML response.

14.3.2 Extending to CSS and JavaScript

By default, `mod_proxy_html` only processes HTML element attributes. If your backend also embeds URLs in inline CSS (`url(...)` values) or JavaScript strings, you'll need:

```
ProxyHTMLExtended On
```

This enables a more aggressive mode that scans for URL-like patterns beyond just HTML attributes. It's slower and has a higher risk of false positives, so only enable it if you actually need it.

14.3.3 Setting the output doctype

`mod_proxy_html` re-serializes the HTML after rewriting, which means it needs to know whether to produce HTML 4, XHTML, or HTML 5. Use `ProxyHTMLDocType` to control this:

```
# Output as HTML 5
ProxyHTMLDocType "<!DOCTYPE html>" HTML

# Or for legacy XHTML
ProxyHTMLDocType "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\"
\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">" XHTML
```

14.3.4 Comparison with mod_substitute

Use `mod_proxy_html` when:

- You need to rewrite URLs in HTML attributes, and you want something that understands HTML structure.
- Your backend produces well-formed HTML and the URLs follow standard patterns.
- You want to avoid accidentally rewriting text content that happens to contain URL-like strings.

Use `mod_substitute` when:

- You need to rewrite content that isn't HTML, or that appears outside of standard HTML attributes (e.g., in `<script>` blocks, JSON responses, plain text).

- You need regex-powered transformations, not just URL mapping.
- The response isn't well-formed HTML and would confuse a parser.

In a complex reverse proxy setup, you may end up using *both* — `mod_proxy_html` for the structural URL rewriting and `mod_substitute` for the edge cases it doesn't cover.

14.4 Filters

All three modules above are implemented as httpd output filters. To use them effectively, it helps to understand what that means.

When httpd generates or receives a response, the content doesn't go directly to the client. Instead, it passes through a chain of *filters* — a pipeline of modules that can inspect, transform, or replace the content as it flows through. You've already been using this concept implicitly: `mod_deflate` (gzip compression) is a filter, `mod_ssl` (encryption) is a filter, and the content transformation modules in this chapter are filters.

14.4.1 Adding filters to the chain

The most common way to add a filter is `AddOutputFilterByType`, which applies the filter only to responses of a specific MIME type:

```
AddOutputFilterByType SUBSTITUTE text/html
AddOutputFilterByType DEFLATE text/html text/css application/javascript
```

You can also use `SetOutputFilter` to apply a filter unconditionally to everything in a given context:

```
<Location "/api">
    SetOutputFilter Sed
</Location>
```

But this is rarely what you want — applying text transformations to binary content will corrupt it.

14.4.2 Filter ordering

Filters run in a defined order. Content transformation filters (`SUBSTITUTE`, `Sed`, `proxy-html`) run before compression (`DEFLATE`), which runs before encryption (`SSL`). This means your substitutions operate on the uncompressed, unencrypted content — which is what you want.

If you stack multiple content transformation filters, they run in the order they were added. A `Substitute` applied before an `OutputSed` will see the original content; the `OutputSed` will see the already-substituted content.

14.4.3 Conditional filtering with `mod_filter`

`AddOutputFilterByType` is good enough for most cases, but `mod_filter` gives you much finer control. You can apply a filter based on request headers, environment variables, response headers, or content type — and you can build conditional pipelines:

```
FilterDeclare rewrite_urls CONTENT_SET
FilterProvider rewrite_urls SUBSTITUTE "%{Content_Type} =~ /text\/html/"
FilterChain    rewrite_urls
```

This declares a filter called `rewrite_urls`, tells `httpd` to use the `SUBSTITUTE` provider when the content type is `text/html`, and adds it to the filter chain. The `FilterProvider` directive can test against any of the standard `httpd` variables, giving you conditional logic that `AddOutputFilterByType` alone can't express.

14.4.4 The escape hatch: `mod_ext_filter`

When none of the built-in filter modules do what you need, `mod_ext_filter` lets you pipe the response body through an arbitrary external program. Define the filter, then add it to the chain:

```
ExtFilterDefine fix-encoding cmd="/usr/bin/iconv -f ISO-8859-1 -t UTF-8"

<Location "/legacy-app">
    SetOutputFilter fix-encoding
</Location>
```

This is powerful — you can use any command-line tool — but it comes at a cost. Every request that triggers the filter forks a new process, which is slow and resource-intensive under load. Use `mod_ext_filter` as a last resort: for prototyping, for low-traffic endpoints, or when the transformation is truly impossible with the built-in modules.

14.4.5 Putting it all together

The key insight for this book's narrative: `mod_rewrite` transforms the *request* — the URL, headers, and environment variables that determine what content the server will produce or fetch. The filter modules discussed in this chapter transform the *response* — the actual content that the client receives.

They're complementary tools. A typical reverse proxy configuration might use `mod_rewrite` (or `ProxyPass`) to route requests to the correct backend, `ProxyPassReverse` to fix response headers, and then `mod_substitute` or `mod_proxy_html` to fix URLs embedded in the response body. Understanding both sides of the request-response lifecycle gives you full control over how your server behaves.

Not to know that no space of regret can make amends
for one life's opportunity misused!

—Charles Dickens, *A Christmas Carol*

In this chapter, we'll present various common problems, and a variety of ways to solve them using `mod_rewrite`, or one of the other tools discussed in this book.

Some of these recipes have already been presented in other parts of the book, but are gathered here to make it easier to find them. We'll also expand, in detail, how they work, and when you might want to use one solution versus another.

Many of these recipes are drawn from questions that appear regularly on the `users@httpd.apache.org` mailing list. They represent real-world problems that real administrators face every day. Where a question comes up repeatedly on the mailing list, we note that—it's a signal that the existing documentation could do a better job of explaining the solution.

15.1 Common Redirects

These are the bread and butter of URL manipulation—the redirects that every web administrator will need at some point.

15.1.1 Redirecting HTTP to HTTPS

Problem: You want to force all traffic to use HTTPS. This is by far the most common question on the `httpd` users mailing list, appearing in dozens of threads over the years. Users frequently struggle with where to put the redirect rules, especially when virtual hosts are involved.

Approach: `Redirect` (preferred), or `mod_rewrite`, or `<If>`

A common pitfall, seen in threads like “Virtual Host - Port 80 to 443,” is putting SSL directives and rewrite rules in the same `<VirtualHost>` block. The correct pattern is to use *two* virtual host blocks: one for port 80 that does nothing but redirect, and one for port 443 that holds the actual site configuration.

The simplest and clearest approach uses two `<VirtualHost>` blocks and a single `Redirect` directive:

```
# Port 80: redirect everything to HTTPS
<VirtualHost *:80>
```

(continues on next page)

(continued from previous page)

```

    ServerName www.example.com
    Redirect permanent / https://www.example.com/
</VirtualHost>

# Port 443: the real site
<VirtualHost *:443>
    ServerName www.example.com
    SSLEngine on
    SSLCertificateFile /etc/pki/tls/certs/example.com.crt
    SSLCertificateKeyFile /etc/pki/tls/private/example.com.key
    DocumentRoot /var/www/html
</VirtualHost>

```

The `Redirect permanent` on port 80 sends a 301 for every request, preserving the original path and query string. The client's browser will cache this redirect, so subsequent visits go straight to HTTPS.

Warning

Do **not** put `SSLEngine on` and `Redirect` in the same `<VirtualHost>` block. The port-80 block handles plaintext HTTP; the port-443 block handles TLS. Mixing them is the single most common mistake seen on the mailing list.

If you need `mod_rewrite` for this (perhaps because you're in a `.htaccess` file and can't define virtual hosts):

```

RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]

```

The `RewriteCond %{HTTPS} off` ensures this only fires for plaintext connections, preventing a redirect loop. See [Chapter 7](#) for details on `RewriteCond`.

Starting with `httpd 2.4`, you can also use an `<If>` expression inside the port-80 virtual host:

```

<If "%{HTTPS} == 'off'">
    Redirect permanent / https://www.example.com/
</If>

```

The `<If>` approach reads more naturally than `RewriteCond`, but the two-`VirtualHost` pattern with a bare `Redirect` remains the cleanest solution. The `<If>` form is most useful when you cannot separate the configuration into two virtual host blocks.

Another common mistake: using `_default_:443` as the virtual host address. This creates a catch-all SSL host that matches *any* hostname, which can cause certificate mismatch warnings if you have multiple domains. Always use an explicit `ServerName` in your SSL virtual host.

15.1.2 Canonicalizing the Hostname (www vs. non-www)

Problem: You want `www.example.com` and `example.com` to resolve to a single canonical URL, to avoid duplicate content in search engines. This comes up frequently on the mailing list, often intertwined with the HTTP-to-HTTPS redirect question.

Approach: Separate `<VirtualHost>` blocks (preferred), or `mod_rewrite` with `RewriteCond %{HTTP_HOST}`

As noted in the “redirect vs. rewrite” thread on the `httpd` users list, the recommended approach from experienced responders is to use separate virtual hosts for hostname canonicalization, rather than `RewriteCond`. This keeps the configuration clearer and avoids accidental interactions with other rewrite rules.

The cleanest approach uses separate `<VirtualHost>` blocks — one canonical, one that redirects:

```
# Redirect non-www to www
<VirtualHost *:443>
  ServerName example.com
  SSLEngine on
  SSLCertificateFile /etc/pki/tls/certs/example.com.crt
  SSLCertificateKeyFile /etc/pki/tls/private/example.com.key
  Redirect permanent / https://www.example.com/
</VirtualHost>

# Canonical host
<VirtualHost *:443>
  ServerName www.example.com
  SSLEngine on
  SSLCertificateFile /etc/pki/tls/certs/example.com.crt
  SSLCertificateKeyFile /etc/pki/tls/private/example.com.key
  DocumentRoot /var/www/html
</VirtualHost>
```

To redirect in the *other* direction (`www` → `non-www`), swap the `ServerName` values.

The separate-`VirtualHost` approach is preferred because:

- It makes the intent obvious to anyone reading the config.
- The redirect `VirtualHost` contains no `DocumentRoot`, no rewrite rules, and no application config — just a single `Redirect`.
- There’s no risk of rewrite-rule interactions.

If you can’t use separate virtual hosts (e.g., you’re in `.htaccess`), use `RewriteCond`:

```
RewriteEngine On
RewriteCond %{HTTP_HOST} !^www\. [NC]
RewriteRule ^ https://www.%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

Note

The [NC] (no-case) flag handles mixed-case hostnames. Without it, Example.COM would not match.

To combine hostname canonicalization with the HTTPS redirect, put them in order — the HTTPS redirect first, then the hostname redirect:

```
# In .htaccess or a single VirtualHost:
RewriteEngine On

# Step 1: Force HTTPS
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]

# Step 2: Force www
RewriteCond %{HTTP_HOST} !^www\. [NC]
RewriteRule ^ https://www.%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

This results in at most two redirects for `http://example.com/page`: first to `https://example.com/page`, then to `https://www.example.com/page`. If you want a single redirect, combine the conditions in the VirtualHost approach — the port-80 redirect points directly to the canonical `https://www.` URL.

15.1.3 Adding or Removing Trailing Slashes

Problem: You want consistent URLs—either always with a trailing slash or always without. `mod_dir`'s `DirectorySlash` directive interacts with this in ways that confuse many users.

Approach: `mod_dir / DirectorySlash`, or `mod_rewrite`

A mailing list thread on “Limiting redirects with `rewriterule/rewritecond`” discusses combining trailing-slash removal with other rewrites to reduce the number of redirects a client experiences. One respondent notes: “be careful about not creating loops, especially if using `.htaccess` files.”

Adding a trailing slash is the default behavior of `mod_dir`. When a request for `/about` matches a directory on disk, `mod_dir` automatically redirects to `/about/`. The `DirectorySlash` directive controls this:

```
# Default behavior - adds trailing slash to directories
DirectorySlash On
```

If you want to *enforce* trailing slashes on all URLs (not just directories), use `mod_rewrite`:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_URI} !(.*)/$
RewriteRule ^(.*)$ /$1/ [R=301,L]
```

The `!-f` condition prevents adding a slash to actual files (you don't want `/style.css/`).

Removing trailing slashes requires disabling `DirectorySlash` and handling it yourself:

```
DirectorySlash Off
```

```
RewriteEngine On
```

```
# Remove trailing slash (except for root /)
```

```
RewriteCond %{REQUEST_FILENAME} !-d
```

```
RewriteRule ^(.+)/$ /$1 [R=301,L]
```

⚠ Warning

Setting `DirectorySlash Off` means `mod_dir` will no longer automatically redirect `/about` to `/about/`, which can cause relative links within that directory to break. You must ensure your application generates absolute URLs or handles this itself.

Avoiding redirect loops: In `.htaccess`, the URI is re-evaluated after each internal rewrite. A rule that adds a slash can interact with `mod_dir`'s own slash-adding logic, creating a loop. The safest pattern is:

```
# In .htaccess - add slash without looping
```

```
RewriteEngine On
```

```
RewriteCond %{REQUEST_FILENAME} -d
```

```
RewriteCond %{REQUEST_URI} !/$
```

```
RewriteRule ^ %{REQUEST_URI}/ [R=301,L]
```

The `-d` check ensures the rule only fires for actual directories, and the `!/$` condition ensures it doesn't fire if the slash is already present. See [Chapter 11](#) for more on avoiding loops.

15.1.4 Redirecting an Entire Site to a New Domain

Problem: You've moved your site to a new domain and want to redirect all old URLs to the new domain, preserving the path. This comes up frequently on the mailing list—a thread on “rewrite in `.htaccess`” shows a user migrating a WordPress site who gets partial redirects because their rewrite rules are in the wrong order.

Approach: Redirect (preferred for simple cases), or `mod_rewrite`

The key mistake in the mailing list thread: placing the domain-migration rewrite rules *after* the WordPress `.htaccess` rules, which short-circuit with `[L]` before the migration rules are reached.

The simplest approach is a single `Redirect` directive:

```
<VirtualHost *:80>
  ServerName old.example.com
  Redirect permanent / https://new.example.com/
</VirtualHost>
```

This redirects every request while preserving the path. `/blog/my-post?id=42` becomes `https://new.example.com/blog/my-post?id=42`.

The `mod_rewrite` equivalent:

```
<VirtualHost *:80>
  ServerName old.example.com
  RewriteEngine On
  RewriteRule ^ https://new.example.com%{REQUEST_URI} [R=301,L]
</VirtualHost>
```

Use the `mod_rewrite` version when you need to add conditions — for example, redirecting only certain paths or excluding an API endpoint from the redirect.

Rule ordering with CMS .htaccess files: If you’re migrating a WordPress (or similar CMS) site, the CMS .htaccess typically contains:

```
# WordPress default
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

If you add your domain-migration redirect *after* these rules, the `[L]` flag on the WordPress rules will stop processing before your redirect is ever reached. Place domain-migration rules **before** any CMS rules:

```
# Domain migration - must come FIRST
RewriteEngine On
RewriteRule ^ https://new.example.com%{REQUEST_URI} [R=301,L]

# WordPress rules below (never reached because of the [L] above)
# ...
```

Better yet, put the Redirect in server config rather than .htaccess — it will be processed before any .htaccess rules are even loaded.

15.1.5 Redirecting Individual Pages That Have Moved

Problem: Specific pages have moved to new URLs (site restructure, CMS migration, etc.) and you need 301 redirects for SEO. A thread on “Redirects and rewrites and performance” discusses a site with *10,000* accumulated redirects from years of migrations, raising the question of when performance becomes a concern.

Approach: Redirect / RedirectMatch (preferred), or RewriteRule, or RewriteMap for large numbers of redirects

For a small number of redirects, Redirect directives are simplest. For thousands of redirects, use a RewriteMap with a DBM or text file to avoid loading thousands of individual directives.

For a handful of redirects, use Redirect:

```
Redirect permanent /old-page.html /new-page.html
Redirect permanent /blog/2019/post /archive/2019/post
```

For pattern-based redirects, use `RedirectMatch`:

```
# Redirect all /blog/YYYY/slug to /archive/YYYY/slug
RedirectMatch permanent ^/blog/([0-9]{4})/(.+)$ /archive/$1/$2
```

For **large numbers** of redirects (hundreds or thousands), individual `Redirect` or `RewriteRule` directives become unwieldy and slow. Use a `RewriteMap` instead:

```
# In server config (not .htaccess - RewriteMap can't go there)
RewriteMap redirects "txt:/etc/httpd/conf/redirect-map.txt"

RewriteEngine On
RewriteCond ${redirects:$1} !=""
RewriteRule ^(.+)$ ${redirects:$1} [R=301,L]
```

The map file is a simple two-column text file:

```
# /etc/httpd/conf/redirect-map.txt
/old-page.html /new-page.html
/blog/2019/post /archive/2019/post
/products/widget /shop/widgets
```

For even better performance with thousands of entries, convert the text map to DBM format using `httxt2dbm`:

```
httxt2dbm -i redirect-map.txt -o redirect-map.dbm
```

Then reference the DBM map:

```
RewriteMap redirects "dbm:/etc/httpd/conf/redirect-map.dbm"
```

DBM lookups are $O(1)$ hash-table lookups regardless of map size, while a text file is scanned linearly. For 10,000+ redirects, the difference is significant. See *Chapter 8* for full details on `RewriteMap` types.

15.1.6 Redirecting Wildcard Subdomains

Problem: You need to redirect `*.oldsite.com` to `newsite.com`, possibly preserving certain allowed subdomains. A detailed thread on “Apache Rewrite - Redirect Wildcard Subdomain” shows a user with complex requirements: some wildcard subdomains should redirect to the base domain, while others should be preserved on the new domain.

Approach: `mod_rewrite` with `RewriteCond %{HTTP_HOST}`

This requires `mod_rewrite` because `Redirect` and `RedirectMatch` cannot match against the hostname. The key is getting the `ServerAlias` right (`*.oldsite.com`) and using `RewriteCond` to capture and selectively route subdomain patterns.

First, ensure DNS is configured with a wildcard record (`*.oldsite.com` → your server IP), and that your virtual host accepts wildcard connections:

```
<VirtualHost *:80>
  ServerName  oldsite.com
  ServerAlias *.oldsite.com

  RewriteEngine On
  RewriteRule ^ https://newsite.com%{REQUEST_URI} [R=301,L]
</VirtualHost>
```

This redirects every subdomain (blog.oldsite.com, shop.oldsite.com, etc.) to the base domain newsite.com, preserving the path.

Preserving the subdomain on the new domain requires capturing it from the Host header:

```
<VirtualHost *:80>
  ServerName  oldsite.com
  ServerAlias *.oldsite.com

  RewriteEngine On
  RewriteCond %{HTTP_HOST} ^(.+)\.oldsite\.com$ [NC]
  RewriteRule ^ https://%1.newsite.com%{REQUEST_URI} [R=301,L]
</VirtualHost>
```

Here %1 is the first capture group from the RewriteCond — the subdomain portion.

Selective handling — redirect most subdomains but keep a few:

```
<VirtualHost *:80>
  ServerName  oldsite.com
  ServerAlias *.oldsite.com

  RewriteEngine On
  # Don't redirect mail or api subdomains
  RewriteCond %{HTTP_HOST} !^(mail|api)\.oldsite\.com$ [NC]
  RewriteCond %{HTTP_HOST} ^(.+)\.oldsite\.com$ [NC]
  RewriteRule ^ https://%1.newsite.com%{REQUEST_URI} [R=301,L]
</VirtualHost>
```

The first RewriteCond excludes mail and api; the second captures the subdomain for redirection. Because multiple RewriteCond lines before a single RewriteRule are ANDed by default, both conditions must be true for the rule to fire.

15.2 Clean and Pretty URLs

Making URLs user-friendly and hiding implementation details.

15.2.1 Removing File Extensions (.php, .html)

Problem: You want /about to serve /about.php without the user seeing the .php extension. A thread on “Remove .php extension but still pass it to PHP-FPM” shows this is especially tricky when PHP-FPM is in the mix, because the proxy handler needs to know the actual file path.

Approach: mod_rewrite (with -f check), or MultiViews (content negotiation)

MultiViews (enabled via Options +MultiViews) can handle this without any rewrite rules at all, but its behavior can be surprising and it has performance implications. The mod_rewrite approach gives more control.

MultiViews (simplest): Enable content negotiation, and Apache will automatically serve /about.php when the client requests /about:

```
<Directory /var/www/html>
  Options +MultiViews
</Directory>
```

That’s it — no rewrite rules needed. Apache looks for files matching the requested path with any known extension and serves the best match.

Note

MultiViews can produce unexpected results if you have both about.html and about.php — Apache will choose based on content negotiation headers. It also adds a small overhead because Apache must scan the directory for matching files on every request.

:module:`mod_rewrite` approach (more control):

```
RewriteEngine On

# If the request doesn't match an existing file or directory
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# And the request with .php appended IS a real file
RewriteCond %{REQUEST_FILENAME}.php -f
RewriteRule ^(.+)$ $1.php [L]
```

The -f checks are essential. Without the final RewriteCond, a request for /nonexistent would be rewritten to /nonexistent.php, which also doesn’t exist, producing a confusing 404.

PHP-FPM / ProxyPassMatch consideration: When PHP is handled by PHP-FPM via ProxyPassMatch, the proxy handler needs the .php extension to know which requests to forward:

```
# Typical PHP-FPM proxy config
ProxyPassMatch ^/(.*\.php(/.*?))$ fcgi://127.0.0.1:9000/var/www/html/$1
```

(continues on next page)

(continued from previous page)

```
# Extensionless rewrite (in .htaccess or <Directory>)
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME}.php -f
RewriteRule ^(.+)$ $1.php [L]
```

The rewrite appends `.php` internally, and then `ProxyPassMatch` matches the `.php` extension and forwards to PHP-FPM. The order matters: the rewrite happens first (in `<Directory>` context), then the proxy match is evaluated against the rewritten URI.

15.2.2 Front Controller Pattern (CMS/Framework Routing)

Problem: Your application framework (WordPress, Laravel, Symfony, etc.) uses a front controller pattern where all requests that don't match a real file should be routed to `index.php`. This is the single most common `.htaccess` configuration on the web, and it generates a steady stream of mailing list questions when it doesn't work.

Approach: `mod_rewrite` in `.htaccess`

The standard WordPress `.htaccess` pattern is:

```
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

A thread on “WordPress `.htaccess` rewrite issue between `httpd` versions” documents a rewrite loop that appeared when migrating from `httpd` 2.4.6 to 2.4.51 with PHP-FPM—the `ProxyPassMatch` for PHP files interacted with the `.htaccess` rewrite rules in an unexpected way, causing the rewrite to loop infinitely.

Here is the standard pattern, annotated line by line:

```
# Enable the rewrite engine
RewriteEngine On

# Set the base URL for relative substitutions in .htaccess.
# If your site lives at /blog/, change this to /blog/.
RewriteBase /

# If the request is literally for index.php, stop here.
# The [L] flag means "last rule" - don't process further.
# The - substitution means "don't rewrite, pass through."
RewriteRule ^index\.php$ - [L]

# If the requested file exists on disk, don't rewrite.
```

(continues on next page)

(continued from previous page)

```

RewriteCond %{REQUEST_FILENAME} !-f

# If the requested directory exists on disk, don't rewrite.
RewriteCond %{REQUEST_FILENAME} !-d

# Everything else: rewrite to index.php.
# The "." pattern matches any non-empty URI.
RewriteRule . /index.php [L]

```

Why the ``^index.php\$`` rule? Without it, the rewrite creates a loop. After the last RewriteRule rewrites to /index.php, the .htaccess is re-evaluated against the *new* URI. The first rule matches index.php and stops, breaking the loop.

What breaks when ``AllowOverride`` is wrong: If the server config has AllowOverride None (the default in many distributions), .htaccess files are completely ignored — no errors, no log entries, nothing. The fix:

```

<Directory /var/www/html>
    AllowOverride FileInfo
</Directory>

```

FileInfo is the minimum needed for RewriteRule directives. AllowOverride All also works but grants more than necessary.

What breaks when ``RewriteBase`` is missing: In .htaccess, mod_rewrite strips the directory prefix from the URI before matching, then prepends it back after substitution. RewriteBase tells it what to prepend. If omitted and the site is in a subdirectory, rewrites produce incorrect paths. For a site at http://example.com/blog/, you need RewriteBase /blog/.

The PHP-FPM loop interaction: When PHP runs via ProxyPassMatch:

```

ProxyPassMatch ^/(.*\.php(/.*?))$ fcgi://127.0.0.1:9000/var/www/html/$1

```

A request for /my-page gets rewritten to /index.php by the .htaccess rules, then ProxyPassMatch forwards it to PHP-FPM. PHP-FPM processes it and returns a response. The loop occurs if ProxyPassMatch triggers a *subrequest* that re-enters .htaccess processing. In httpd 2.4.51+, the PT (passthrough) flag behavior changed slightly, which can trigger this. The fix: use SetHandler instead of ProxyPassMatch:

```

<FilesMatch "\.php$" >
    SetHandler "proxy:fcgi://127.0.0.1:9000"
</FilesMatch>

```

SetHandler avoids the regex-matching path entirely and doesn't trigger rewrite re-entry.

15.2.3 Mapping Clean URL Paths to Query Parameters

Problem: You want `/products/widget-42` to internally map to `/product.php?id=widget-42`. This is a classic `mod_rewrite` use case and appears in many mailing list threads. A common mistake (seen in “RewriteRule not working, 404 error obtained”) is that `AllowOverride` is not set correctly, so the `.htaccess` rules are silently ignored.

Approach: `mod_rewrite`

Simple one-segment mapping — `/products/shoes` to `/products.php?cat=shoes`:

```
RewriteEngine On
RewriteRule ^products/([a-zA-Z0-9_-]+)$ /products.php?cat=$1 [L]
```

The `$1` backreference captures whatever matched inside the parentheses. See *Chapter 1* for regex details.

Multi-segment paths — `/products/shoes/running` to `/products.php?cat=shoes&sub=running`:

```
RewriteRule ^products/([^/]+)/([^/]+)$ /products.php?cat=$1&sub=$2 [L]
```

With optional segments — `/products/shoes` or `/products/shoes/running`:

```
RewriteRule ^products/([^/]+)/([/^/]+)?$ /products.php?cat=$1&sub=$3 [L]
```

If the second segment is absent, `$3` is empty and the query parameter `sub=` has no value, which the application should handle.

The “AllowOverride” pitfall: If these rules are in `.htaccess` and `AllowOverride` does not include `FileInfo`, the rules are silently ignored. There’s no error message, no log entry — the `.htaccess` file simply has no effect. Enable tracing to confirm rules are being processed:

```
# In server config
<Directory /var/www/html>
    AllowOverride FileInfo
</Directory>

# Temporarily enable rewrite logging
LogLevel alert rewrite:trace3
```

If you see no rewrite log entries at all for a request that should match your `.htaccess` rules, `AllowOverride` is the likely culprit.

 Tip

These patterns assume `mod_rewrite` in `.htaccess`. In server config (`<VirtualHost>` or `<Directory>`), the URI includes the leading slash, so the pattern becomes `^/products/([^/]+)$`. See *Chapter 11* for the full list of context differences.

15.3 Access Control

Using URL manipulation for access control purposes. (Note: `mod_rewrite` is generally *not* the best tool for access control—`Require`, `<If>`, and `mod_authz_*` are usually better choices.)

15.3.1 Blocking Hotlinking (Referrer-Based Access)

Problem: Other sites are embedding your images directly, consuming your bandwidth. You want to block or redirect requests that come from other domains.

Approach: `mod_rewrite` with `RewriteCond` `%{HTTP_REFERER}` (traditional), or `<If>` expression (modern, preferred)

:module:`mod_rewrite` approach (traditional):

```
RewriteEngine On
RewriteCond %{HTTP_REFERER} !^$ [NC]
RewriteCond %{HTTP_REFERER} !^https?://(www\.)?example\.com [NC]
RewriteRule \.(jpg|jpeg|png|gif|webp|svg)$ - [F]
```

Line by line:

1. `!^$` — allow empty referrers (direct visits, bookmarks, privacy extensions). Without this, your own users following bookmarks would be blocked.
2. `!^https?://(www\.)?example\.com` — allow requests from your own domain.
3. The `RewriteRule` matches image extensions and returns 403 Forbidden (`[F]`).

To serve a placeholder image instead of a 403:

```
RewriteRule \.(jpg|jpeg|png|gif|webp|svg)$ /images/hotlink-notice.png [L]
```

SetEnvIf + Require approach (modern, preferred):

```
SetEnvIf Referer "^$" local_ref
SetEnvIf Referer "^https?://(www\.)?example\.com" local_ref

<FilesMatch "\.(jpg|jpeg|png|gif|webp|svg)$">
  Require env local_ref
</FilesMatch>
```

This approach is cleaner — the access control logic lives in the authorization layer where it belongs, not in URL rewriting.

⚠ Warning

The Referer header is trivially spoofed and frequently absent. Privacy-focused browsers, extensions, and corporate proxies strip it. Hotlink protection is a deterrent, not a security boundary. Don't use it to protect genuinely sensitive content — use authentication instead.

15.3.2 Blocking Requests by User-Agent

Problem: You want to block specific bots, scrapers, or vulnerability scanners based on their User-Agent string. Several mailing list threads discuss this in the context of “Unknown accepted traffic” and bot mitigation.

Approach: `mod_rewrite` with `RewriteCond` `%{HTTP_USER_AGENT}` (traditional), or `<If> / SetEnvIf` with `Require` (modern, preferred)

SetEnvIf + Require approach (recommended):

```
SetEnvIf User-Agent "BadBot" bad_bot
SetEnvIf User-Agent "EvilScrapper" bad_bot
SetEnvIf User-Agent "VulnScanner" bad_bot

<Directory /var/www/html>
  <If "reqenv('bad_bot') != ''">
    Require all denied
  </If>
</Directory>
```

Or more concisely with `Require`:

```
SetEnvIf User-Agent "BadBot|EvilScrapper|VulnScanner" bad_bot

<Directory /var/www/html>
  Require not env bad_bot
</Directory>
```

:module:`mod_rewrite` approach (traditional):

```
RewriteEngine On
RewriteCond %{HTTP_USER_AGENT} (BadBot|EvilScrapper|VulnScanner) [NC]
RewriteRule ^ - [F]
```

The `SetEnvIf` approach is preferred because:

- It separates *identification* (`SetEnvIf`) from *authorization* (`Require`), making the config easier to read and maintain.
- The `Require` directive integrates with `httpd`'s authorization framework, producing proper 403 responses with correct logging.

- mod_rewrite's [F] flag works, but using the rewrite engine for access control is using the wrong tool for the job.

Note

User-Agent strings are trivially spoofed. Any bot that wants to evade detection can send a browser-like User-Agent. This technique is useful against lazy bots and automated scanners but is not a substitute for rate limiting or WAF rules.

15.3.3 Cookie-Based Redirect to Login Page

Problem: You want to redirect users to a login page if a specific authentication cookie is not present. A thread on “redirects on Apache 2.4” shows a user trying to check for a web_route cookie and redirect unauthenticated users to a login portal.

Approach: mod_rewrite with RewriteCond %{HTTP_COOKIE}

The common mistake from the mailing list: the rewrite rules are in the wrong order, and the [L] flag on the front controller rule prevents the cookie check from ever being evaluated.

Check for the absence of an authentication cookie and redirect:

RewriteEngine On

```
# Don't redirect the login page itself (avoid loop)
RewriteCond %{REQUEST_URI} !^/login
# Don't redirect static assets
RewriteCond %{REQUEST_URI} !^(css|js|images)/

# Check if the auth cookie is missing
RewriteCond %{HTTP_COOKIE} !auth_token=
RewriteRule ^ /login?redirect=%{REQUEST_URI} [R=302,L]
```

Key points:

- The !^/login exclusion prevents a redirect loop — without it, the login page itself would trigger another redirect.
- Static asset exclusions prevent CSS and images from being inaccessible on the login page.
- Use [R=302] (temporary), not [R=301] (permanent). A permanent redirect gets cached by the browser, so even after the user logs in and obtains the cookie, the browser may still redirect to /login.
- The %{REQUEST_URI} in the query string passes the original URL to the login handler, enabling redirect-after-login.

Rule ordering matters. If you combine this with a front controller pattern, the cookie check must come *before* the front controller rules:

```
RewriteEngine On

# 1. Cookie check (redirect to login)
RewriteCond %{REQUEST_URI} !^/login
RewriteCond %{REQUEST_URI} !^(css|js|images)/
RewriteCond %{HTTP_COOKIE} !auth_token=
RewriteRule ^ /login?redirect=%{REQUEST_URI} [R=302,L]

# 2. Front controller (send everything else to index.php)
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

If the front controller rules come first, their [L] flag stops processing and the cookie check never runs.

 **Tip**

This pattern is a lightweight guard, not a security mechanism. The cookie can be forged, and the check runs on every request. For real authentication, use `mod_auth_form` (httpd's built-in form-based auth), `mod_auth_openidc` (for OAuth2/OIDC), or handle authentication in your application or reverse proxy layer.

15.3.4 IP-Based Access Control

Problem: You want to restrict access to certain paths based on client IP address.

Approach: Require `ip` (strongly preferred), `<If>` expressions, or `mod_rewrite` with `RewriteCond` `%{REMOTE_ADDR}` (not recommended)

Require ip (strongly preferred):

```
<Location /admin>
  Require ip 10.0.0.0/8
  Require ip 192.168.1.0/24
  Require ip 2001:db8::/32
</Location>
```

To allow a specific IP *and* deny everything else:

```
<Location /admin>
  Require ip 10.1.2.3
</Location>
```

To combine IP restrictions with authentication (require *both*):

```
<Location /admin>
  <RequireAll>
```

(continues on next page)

(continued from previous page)

```

    Require ip 10.0.0.0/8
    Require valid-user
</RequireAll>
</Location>

```

<If> expression approach:

```

<If "%{REMOTE_ADDR} -ipmatch '10.0.0.0/8'">
    # Allowed
</If>
<Else>
    Require all denied
</Else>

```

:module:`mod_rewrite` approach (do not use for this):

```

# This works, but don't do it
RewriteEngine On
RewriteCond %{REMOTE_ADDR} !^10\.
RewriteRule ^/admin - [F]

```

This is worse than `Require ip` in every way:

- The regex `!^10\.` is an approximation of `10.0.0.0/8` — it doesn't actually do CIDR matching, so it's easy to get wrong.
- It bypasses the authorization framework, so `Require` directives in the same scope may not behave as expected.
- It doesn't log the denial through the standard authorization log.
- It doesn't support IPv6 CIDR notation.

Use `Require ip`. It exists precisely for this purpose, handles CIDR correctly for both IPv4 and IPv6, integrates with the authorization framework, and is far easier to read and audit.

15.4 Proxying

Rewriting in the context of reverse proxying is a common source of confusion, as the mailing list amply demonstrates.

15.4.1 Rewriting URLs for a Reverse Proxy Backend

Problem: Your reverse proxy needs to strip or add a path prefix when forwarding requests to a backend application. A detailed thread on “`mod_proxy_http` rewrite problem” shows a user struggling with balancer configuration where the rewrite rule incorrectly strips the application context path, causing authentication to break.

Approach: ProxyPass path mapping (preferred), or `mod_rewrite` with `[P]` flag

The recommended approach is to let ProxyPass and ProxyPassReverse handle the path mapping. Using mod_rewrite with [P] should be a last resort, because it bypasses the connection pooling and other optimizations of mod_proxy.

ProxyPass path mapping (preferred):

```
# Forward /app/ to a backend running on port 8080 at /
ProxyPass      /app/ http://backend.local:8080/
ProxyPassReverse /app/ http://backend.local:8080/
```

ProxyPass maps the incoming path to the backend path. ProxyPassReverse rewrites Location headers in the backend's responses so that redirects issued by the backend (e.g., Location: http://backend.local:8080/login) are translated back to the client-facing URL (/app/login).

Stripping a prefix:

```
# Client requests /api/v2/users
# Backend expects /v2/users (no /api prefix)
ProxyPass      /api/ http://backend.local:8080/
ProxyPassReverse /api/ http://backend.local:8080/
```

The path mapping in ProxyPass handles the prefix stripping automatically — /api/v2/users becomes /v2/users on the backend.

:module:`mod_rewrite` with [P] flag (last resort):

```
RewriteEngine On
RewriteRule ^/app/(.*)$ http://backend.local:8080/$1 [P]
ProxyPassReverse /app/ http://backend.local:8080/
```

Warning

The [P] flag forces the request through mod_proxy, but it **bypasses** ProxyPass's connection pooling and worker configuration. Each [P] request creates a new connection to the backend. For high-traffic sites, this is significantly less efficient.

You still need ProxyPassReverse even when using [P] — the flag handles the *request* path but not the *response* headers.

Do not mix ProxyPass and RewriteRule [P] for the same path:

```
# WRONG - these will conflict
ProxyPass /app/ http://backend.local:8080/
RewriteRule ^/app/special/(.*)$ http://other-backend:8080/$1 [P]
```

ProxyPass is processed before RewriteRule in most contexts, so the rewrite rule may never be reached. If you need conditional proxying, use RewriteRule [P] for *all* paths in that scope, or use ProxyPass with <Location> blocks and <If> conditions.

15.4.2 Redirects Behind a TLS-Terminating Proxy

Problem: Your httpd sits behind a load balancer or CDN that terminates TLS. The `HTTPS` variable is always `off` from httpd's perspective, causing redirect loops when you try to force HTTPS. A thread on "Configuring redirects httpd behind a TLS-terminating proxy" discusses this exact scenario.

Approach: `mod_rewrite` with `RewriteCond HTTPS off` or `<If>` with `req('X-Forwarded-Proto')`

When httpd sits behind a load balancer or CDN that terminates TLS, the connection between the proxy and httpd is plain HTTP. From httpd's perspective, `HTTPS` is always `off`. The standard "force HTTPS" redirect creates an infinite loop:

```
# This loops behind a TLS-terminating proxy!
RewriteCond HTTPS off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

The fix: check the `X-Forwarded-Proto` header set by the proxy instead:

```
RewriteEngine On
RewriteCond %{HTTP:X-Forwarded-Proto} =http
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [R=301,L]
```

Or with an `<If>` expression (cleaner):

```
<If "req('X-Forwarded-Proto') == 'http'">
  Redirect permanent / https://www.example.com/
</If>
```

Using `module:mod_remoteip` to make `HTTPS` work correctly:

`mod_remoteip` can be configured to trust the proxy's forwarded headers, allowing the standard `HTTPS` variable to reflect the client's actual connection:

```
# Trust the proxy at 10.0.0.0/8
RemoteIPHeader X-Forwarded-For
RemoteIPTrustedProxy 10.0.0.0/8
```

Note that `mod_remoteip` handles the client IP (`X-Forwarded-For`), not the protocol. For protocol detection, you still need to check `X-Forwarded-Proto` explicitly. Some setups use `RequestHeader set X-Forwarded-Proto "https"` on the proxy side and then check it on the httpd side.

Warning

Only trust `X-Forwarded-Proto` from known proxies. If a client sends this header directly (bypassing the proxy), they can trick httpd into thinking the connection is secure. Use firewall rules or `<If>` conditions to ensure only your proxy can set this header.

15.4.3 WebSocket Proxying

Problem: Your application uses WebSockets and you need to proxy `ws://` or `wss://` traffic through `httpd`. A recurring thread on “Web sockets & proxy_pass - No protocol handler was valid for the URL” shows users struggling to get `mod_proxy_wstunnel` working.

Approach: `mod_proxy_wstunnel` with `ProxyPass`, sometimes combined with `mod_rewrite` for upgrade detection

Basic WebSocket proxy with `:module:`mod_proxy_wstunnel`:`

```
# Enable required modules
LoadModule proxy_module          modules/mod_proxy.so
LoadModule proxy_wstunnel_module modules/mod_proxy_wstunnel.so

# Proxy WebSocket connections at /ws/ to the backend
ProxyPass          /ws/ ws://backend.local:8080/ws/
ProxyPassReverse  /ws/ ws://backend.local:8080/ws/
```

For secure WebSockets (`wss://`), the TLS termination happens at `httpd`; the backend connection can remain plain `ws://`:

```
<VirtualHost *:443>
    SSLEngine on
    # ... SSL config ...

    # Client connects via wss://, httpd proxies as ws://
    ProxyPass          /ws/ ws://backend.local:8080/ws/
    ProxyPassReverse  /ws/ ws://backend.local:8080/ws/
</VirtualHost>
```

Upgrade detection with `:module:`mod_rewrite`` — for applications where the same URL handles both HTTP and WebSocket (e.g., `Socket.IO`):

```
RewriteEngine On
RewriteCond %{HTTP:Upgrade} websocket [NC]
RewriteRule ^/app/(.*)$ ws://backend.local:8080/$1 [P,L]

# Non-WebSocket requests go through normal HTTP proxy
ProxyPass          /app/ http://backend.local:8080/
ProxyPassReverse  /app/ http://backend.local:8080/
```

The `RewriteCond` checks for the `Upgrade: websocket` header that initiates the WebSocket handshake. Only those requests are routed to the `ws://` backend; everything else goes through the normal `ProxyPass`.

Common pitfalls:

1. “No protocol handler was valid for the URL” — `mod_proxy_wstunnel` is not loaded. Add `LoadModule proxy_wstunnel_module`.
2. **Timeout disconnects** — WebSocket connections are long-lived. Set a higher timeout:

```
ProxyTimeout 600
ProxyPass /ws/ ws://backend.local:8080/ws/ timeout=600
```

3. **httpd 2.4.47+** added `ProxyWebsocketFallbackToProxyHttp` which allows `mod_proxy_http` to handle WebSocket upgrades directly, without `mod_proxy_wstunnel`. If you're on a recent version:

```
ProxyPass /ws/ http://backend.local:8080/ws/ upgrade=websocket
```

The `upgrade=websocket` parameter tells `mod_proxy_http` to handle the `Upgrade` header and tunnel the connection.

15.5 Query String Manipulation

Rewriting query strings requires special techniques because the query string is not part of the URL path that `RewriteRule` matches against.

15.5.1 Capturing and Rewriting Query Strings

Problem: You need to rewrite based on query string parameters, or transform query strings during a redirect. A thread on “Mod_rewrite too many redirects” shows a user trying to redirect `/?1234ab` to `/welcome?trackFor=0&trackNo=1234ab` and getting a redirect loop.

Approach: `mod_rewrite` with `RewriteCond` `%{QUERY_STRING}` and `[QSA]` or `[QSD]` flags

The solution from the mailing list: the original `RewriteRule` `^(.*)$` pattern was matching the redirect *target* as well, causing an infinite loop. Changing to `^/$` (matching only the root) fixed the loop.

`RewriteRule` matches only the URL path—it never sees the query string. To match or capture query string parameters, use `RewriteCond` `%{QUERY_STRING}`:

```
# Redirect /?code=ABC123 to /welcome?track=ABC123
RewriteEngine On
RewriteCond %{QUERY_STRING} ^code=([a-zA-Z0-9]+)$
RewriteRule ^/?$ /welcome?track=%1 [R=301,L]
```

Here `%1` is a backreference to the first capture group in the `RewriteCond` pattern (not `$1`, which refers to the `RewriteRule` pattern). See *Chapter 1* for the full backreference syntax.

The `[QSA]` flag (Query String Append): by default, if the `RewriteRule` substitution contains a query string, it *replaces* the original. `[QSA]` appends the original query string to the new one instead:

```
# /products/widget?color=red
# Without QSA -> /catalog.php?item=widget (color=red is lost)
# With QSA    -> /catalog.php?item=widget&color=red

RewriteRule ^/products/(.+$) /catalog.php?item=$1 [QSA,L]
```

The `[QSD]` flag (Query String Discard): removes the query string entirely from the rewritten URL. See the next recipe for details.

Avoiding redirect loops with query strings: The most common loop occurs when the RewriteRule pattern is too broad:

```
# BUG: ^(.*)$ matches /welcome too, causing a loop
RewriteCond %{QUERY_STRING} ^code=(.+)
RewriteRule ^(.*)$ /welcome?track=%1 [R=301,L]

# FIX: match only the specific source URL
RewriteCond %{QUERY_STRING} ^code=(.+)
RewriteRule ^/?$ /welcome?track=%1 [R=301,L]
```

Another anti-loop technique is to add a condition that checks whether the rewrite has already happened:

```
RewriteCond %{QUERY_STRING} ^code=(.+)
RewriteCond %{QUERY_STRING} !track= # don't re-rewrite
RewriteRule ^/?$ /welcome?track=%1 [R=301,L]
```

15.5.2 Stripping Query Strings

Problem: You want to remove query strings from URLs, either for SEO cleanliness or to prevent parameter injection, but you need to preserve query strings on certain specific URLs. A thread on “Stripping query string except from specific URL” shows this exact use case.

Approach: mod_rewrite with [QSD] flag and RewriteCond exceptions

Blanket query string removal with [QSD] (Query String Discard, available since httpd 2.4.0):

```
# Strip query strings from all requests
RewriteEngine On
RewriteCond %{QUERY_STRING} .
RewriteRule ^ %{REQUEST_URI} [QSD,R=301,L]
```

The RewriteCond ensures this only fires when a query string is actually present, avoiding a redirect loop on requests that already have no query string.

Excluding specific paths from query string stripping:

```
RewriteEngine On
# Don't strip query strings from the search page or API
RewriteCond %{REQUEST_URI} !^/search
RewriteCond %{REQUEST_URI} !^/api/
RewriteCond %{QUERY_STRING} .
RewriteRule ^ %{REQUEST_URI} [QSD,R=301,L]
```

The pre-2.4 method (the trailing ? trick): Before [QSD] existed, you discarded the query string by appending a bare ? to the substitution target:

```
# Old method: trailing ? discards the original query string
RewriteRule ^/old-page$ /new-page? [R=301,L]
```

This works because the ? starts a new (empty) query string, replacing the original. It still works in 2.4+, but [QSD] is clearer about intent:

```
# Equivalent, but more readable
RewriteRule ^/old-page$ /new-page [QSD,R=301,L]
```

Stripping only specific parameters (keeping the rest):

```
# Remove the 'fbclid' tracking parameter, keep everything else
RewriteEngine On
RewriteCond %{QUERY_STRING} ^(.*)?(?:^|&)fbclid=[^&]*(.*)$
RewriteRule ^ %{REQUEST_URI}?%1%2 [R=301,L]
```

This is fiddly regex work. For stripping multiple tracking parameters (utm_*, fbclid, gclid), consider whether a RewriteMap prg: script would be more maintainable than a wall of regex.

15.5.3 Using SetEnvIf with Query Strings

Problem: You need to set environment variables or control logging based on query string parameters. A thread on “Using SetEnvIf for query string” shows a user trying to conditionally set environment variables.

Approach: SetEnvIf with QUERY_STRING variable, or mod_rewrite with [E=VAR:value] flag

SetEnvIf can match against Query_String (note the underscore) to set environment variables without involving mod_rewrite at all:

```
# Set an environment variable when a debug parameter is present
SetEnvIf Query_String "debug=true" IS_DEBUG

# Suppress logging for health-check requests with ?ping
SetEnvIf Query_String "^ping$" no_log

# Use the no_log variable to exclude from access log
CustomLog /var/log/httpd/access_log combined env=!no_log
```

Conditional cache headers based on query string:

```
# Don't cache URLs with a session ID in the query string
SetEnvIf Query_String "sid=" NO_CACHE
Header set Cache-Control "no-store" env=NO_CACHE
```

:module:`mod_rewrite` [E=] flag — set environment variables during rewrite processing:

```
# Tag requests with a tracking parameter
RewriteEngine On
RewriteCond %{QUERY_STRING} utm_source=([^&]+)
RewriteRule ^ - [E=TRACKING_SOURCE:%1]

# Use the variable in a log format
```

(continues on next page)

(continued from previous page)

```
# In server config:
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{TRACKING_SOURCE}e\"" tracking
CustomLog /var/log/httpd/tracking.log tracking env=TRACKING_SOURCE
```

The `-` target means “don’t rewrite the URL”—just set the variable and move on.

Note

In `.htaccess` context, environment variables set by `[E=]` are prefixed with `REDIRECT_` after the rewrite completes. So `TRACKING_SOURCE` becomes `REDIRECT_TRACKING_SOURCE`. This catches many people off guard. To access it in later directives, use `%{ENV:REDIRECT_TRACKING_SOURCE}` or set it with `SetEnv` to copy it to a non-prefixed name.

When to use which: Use `SetEnvIf` when you just need to set a variable based on a simple pattern match—it’s faster and clearer. Use `[E=]` when you’re already writing rewrite rules and need to capture part of the query string into a variable using backreferences.

15.6 Edge Cases and Gotchas

These recipes address the tricky situations that generate the most confused questions on the mailing list.

15.6.1 Diagnosing and Fixing Rewrite Loops

Problem: Your rewrite rules produce “too many redirects” errors or infinite internal loops. This is the single most common class of rewrite problem on the mailing list. Threads include “Mod_rewrite too many redirects,” “SSO Kerberos REMOTE_USER RewriteRule Endless Loop,” and “WordPress .htaccess rewrite issue between httpd versions.”

Approach: `RewriteLog / LogLevel rewrite:trace`, plus rule design patterns to break loops

The Kerberos/SSO thread shows a particularly interesting edge case: a `RewriteCond %{LA-U:REMOTE_USER}` rule that works for 97% of users but creates an endless loop for the other 3%, due to special characters in certain usernames interacting with the lookahead mechanism.

Step 1: Enable the rewrite log.

```
LogLevel warn rewrite:trace3
```

Reproduce the “too many redirects” error and examine the error log. You’ll see the rewrite engine processing the same URI repeatedly.

Step 2: Identify the looping rule.

Look for a pattern like this in the log:

```
rewrite 'welcome' -> '/index.php'
rewrite 'index.php' -> '/index.php'
```

(continues on next page)

(continued from previous page)

```
rewrite 'index.php' -> '/index.php'
...
```

The repeated line tells you which rule is firing in a loop and what it's matching.

Step 3: Fix it. The cause is almost always one of these patterns:

Pattern A — Overly broad rule matching its own target:

```
# BUG: ^(.*)$ matches everything, including /welcome itself
RewriteRule ^(.*)$ /welcome [R=301,L]
# Request for /about -> 301 /welcome -> 301 /welcome -> loop!

# FIX: add a condition to exclude the target
RewriteCond %{REQUEST_URI} !^/welcome$
RewriteRule ^(.*)$ /welcome [R=301,L]
```

Pattern B — .htaccess re-processing after rewrite:

```
# BUG: in .htaccess, [L] restarts processing from the top
RewriteRule ^old-page$ /new-page [R=301,L]
# If /new-page also lives under this .htaccess, the rules re-run

# FIX: the rule only matches 'old-page', so /new-page won't match.
# But if you used ^(.*)$, you need a stop condition:
RewriteCond %{ENV:REDIRECT_STATUS} ^$
RewriteRule ^(.*)$ /new-page [R=301,L]
```

The REDIRECT_STATUS variable is empty on the first pass and set to the status code (e.g., 200) on subsequent passes. This is a reliable way to detect re-processing.

Pattern C — Query string rewrite creates a loop:

```
# BUG: rewrites /?code=123 to /welcome?track=123
# then /welcome?track=123 matches ^(.*)$ again
RewriteCond %{QUERY_STRING} ^code=(.+)
RewriteRule ^(.*)$ /welcome?track=%1 [R=301,L]

# FIX: match only the original URL, not the rewritten one
RewriteCond %{QUERY_STRING} ^code=(.+)
RewriteRule ^/?$ /welcome?track=%1 [R=301,L]
```

Pattern D — WordPress/PHP-FPM interaction:

When PHP runs as a ProxyPassMatch backend, the proxy subrequest can re-trigger .htaccess rewrite rules:

```
# In server config
ProxyPassMatch "^/(.*\.php)$" "fcgi://127.0.0.1:9000/var/www/html/$1"
```

(continues on next page)

(continued from previous page)

```
# In .htaccess (WordPress default)
RewriteRule . /index.php [L]
```

The rewrite sends the request to `/index.php`. The `ProxyPassMatch` proxies it to PHP-FPM. But the proxy subrequest re-enters the `.htaccess` processing, and `.` matches `index.php` again.

Fix: add a condition to skip the rewrite if the request is already for a PHP file, or use `SetHandler` instead of `ProxyPassMatch`:

```
<FilesMatch "\.php$">
  SetHandler "proxy:fcgi://127.0.0.1:9000"
</FilesMatch>
```

The nuclear option: `httpd`'s built-in loop detection stops processing after 10 internal redirects (configurable with `LimitInternalRecursion`). If you see `Request exceeded the limit of 10 internal redirects` in the error log, you have a loop.

15.6.2 .htaccess vs. Server Config Context Differences

Problem: Your rewrite rules work in `httpd.conf` but not in `.htaccess` (or vice versa). This is one of the most frequent sources of confusion on the mailing list. In “Rewrite not applied?” a user has rules in server config that are silently not being evaluated, with no log entries even at `rewrite:trace5`.

Approach: Understanding the per-dir context

The key differences:

- In `.htaccess`, the leading slash is stripped from the URI before matching
- `RewriteBase` matters in `.htaccess` but not in server config
- `AllowOverride` must include `FileInfo` for rewrite rules to work in `.htaccess`
- `[L]` in `.htaccess` doesn't truly stop processing—the rewritten URL goes through the entire `.htaccess` again

Side-by-side: the same rewrite in both contexts.

Rewriting `/products/widget` to `/catalog.php?item=widget`:

```
# In server config (httpd.conf or <VirtualHost>)
RewriteEngine On
RewriteRule ^/products/(.+)$ /catalog.php?item=$1 [L]
```

```
# In .htaccess (at document root)
RewriteEngine On
RewriteRule ^products/(.+)$ catalog.php?item=$1 [L]
```

Key difference: in `.htaccess`, the leading slash is stripped. The rewrite engine operates on the path *relative to the directory* containing the `.htaccess` file.

RewriteBase tells mod_rewrite what URL prefix corresponds to the directory containing the .htaccess file. It matters when your .htaccess is in a subdirectory:

```
# .htaccess in /var/www/html/myapp/
# Without RewriteBase, the rewrite target is relative to /
# With it, the target is relative to /myapp/

RewriteEngine On
RewriteBase /myapp/
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
# Resolves to /myapp/index.php, not /index.php
```

Without RewriteBase /myapp/, the rule would produce /index.php (at the document root) instead of /myapp/index.php.

The AllowOverride pitfall: If AllowOverride does not include FileInfo, .htaccess rewrite rules are silently ignored. No error, no log entry, nothing—the rules simply don’t run:

```
# This silently disables ALL .htaccess rewrite rules
<Directory "/var/www/html">
    AllowOverride None
</Directory>

# This enables them
<Directory "/var/www/html">
    AllowOverride FileInfo
</Directory>

# Or allow everything (common but less secure)
<Directory "/var/www/html">
    AllowOverride All
</Directory>
```

Diagnostic tip: If your .htaccess rules do nothing and even LogLevel rewrite:trace8 produces no output for that directory, check AllowOverride first. The rewrite engine isn’t ignoring your rules—it was never invoked.

Performance note: .htaccess files are read on every request. The server walks the directory tree from the document root to the requested file, reading each .htaccess file along the way. For high-traffic sites, putting rules in server config (which is parsed once at startup) is measurably faster.

15.6.3 Rule Ordering and the [L] Flag

Problem: Your rewrite rules aren’t behaving as expected because of ordering issues. The “rewrite in .htaccess” thread shows a user whose domain migration redirect is placed *after* WordPress rewrite rules that include [L]—the migration rules are never reached.

Approach: Understanding rule processing order

How [L] works in server config: It means “stop processing rules now.” The rewritten URL is the final result.

How [L] works in .htaccess: It means “stop processing rules *for this pass*.” The rewritten URL then goes back through the entire .htaccess file from the top. This re-processing continues until no rule matches, or until mod_rewrite’s internal redirect limit (10 by default) is reached.

This is the single most common source of confusion in mod_rewrite.

Example: The WordPress .htaccess pattern:

```
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

Here’s what happens with a request for /about:

1. **Pass 1:** /about doesn’t match `^index\.php$`. It does match `.` (with the conditions satisfied). Rewritten to `/index.php`. `[L]` stops this pass.
2. **Pass 2:** `/index.php` matches `^index\.php$`. The `-` target means “don’t rewrite.” `[L]` stops this pass.
3. **Pass 3:** No rule changes the URI. Processing ends.

The `^index\.php$ - [L]` rule exists solely to break the re-processing loop. Without it, `.` would match `index.php` again, rewriting it to `/index.php...` which would match `.` again, and so on until the redirect limit.

Rule processing order between server config and .htaccess:

1. Server config (`httpd.conf / <VirtualHost>`) rules run first
2. Then per-directory .htaccess rules run
3. `[L]` in server config does *not* prevent .htaccess rules from running

RewriteCond binds only to the immediately following RewriteRule:

```
# WRONG: this condition does NOT apply to both rules
RewriteCond %{HTTP_HOST} ^www\.example\.com$
RewriteRule ^/foo /bar [L]
RewriteRule ^/baz /qux [L]      # No condition! Matches all hostnames.

# CORRECT: repeat the condition for each rule
RewriteCond %{HTTP_HOST} ^www\.example\.com$
RewriteRule ^/foo /bar [L]
RewriteCond %{HTTP_HOST} ^www\.example\.com$
RewriteRule ^/baz /qux [L]
```

This surprises people who expect conditions to act like `if` blocks in programming languages. Each `RewriteCond` is consumed by the first `RewriteRule` that follows it.

15.6.4 Debugging Rewrite Rules with the Rewrite Log

Problem: You can't figure out why your rewrite rules aren't doing what you expect. The thread on “Redirects and rewrites and performance” shows a user asking how to trace a *specific* redirect without overwhelming the log with data from the entire site.

Approach: `LogLevel rewrite:trace1` through `rewrite:trace8`

Enable the rewrite log by setting `LogLevel` for the rewrite module:

```
# Useful levels:
# rewrite:trace1 --- shows which rules match
# rewrite:trace2 --- adds rewriting result
# rewrite:trace3 --- adds condition evaluation (the sweet spot)
# rewrite:trace4-8 --- increasingly verbose internals

LogLevel warn rewrite:trace3
```

Warning

Never leave `rewrite:trace3` or higher enabled in production. It generates enormous amounts of log data—one entry per condition per rule per request. Enable it temporarily for debugging, then turn it off.

Reading the log output: At `trace3`, each request produces entries like this:

```
[rewrite:trace3] [pid 1234] mod_rewrite.c(475): [client 10.0.0.1:54321]
 10.0.0.1 - - [perdir /var/www/html/] strip per-dir prefix: /var/www/html/about.
↪-> about
[rewrite:trace3] [pid 1234] mod_rewrite.c(475): [client 10.0.0.1:54321]
 10.0.0.1 - - [perdir /var/www/html/] applying pattern '^index\.php$' to uri
↪'about'
[rewrite:trace2] [pid 1234] mod_rewrite.c(475): [client 10.0.0.1:54321]
 10.0.0.1 - - [perdir /var/www/html/] rewrite 'about' -> '/index.php'
```

Key things to look for:

- `strip per-dir prefix` — confirms you're in per-directory (`.htaccess`) context
- `applying pattern` — shows which rule is being tested against what URI
- `rewrite '...' -> '...'` — the actual rewrite result
- If you see the *same* URI being rewritten repeatedly, you have a loop

Filtering the log to focus on specific URLs:

You can set the log level per-directory or per-location:

```
# Only trace rewrites for the /api/ path
<Location "/api/">
  LogLevel warn rewrite:trace3
</Location>
```

Or filter the log file after the fact:

```
# Show only entries for a specific URI
grep 'about-us' /var/log/httpd/error_log | grep rewrite

# Show only the rewrite results (not every condition check)
grep 'rewrite:trace2' /var/log/httpd/error_log

# Watch in real time
tail -f /var/log/httpd/error_log | grep rewrite
```

Tip: When debugging in `.htaccess`, remember that `[L]` does not stop processing—it restarts the rule set. So you’ll see the same URI processed multiple times in the log. This is normal behavior, not a bug. See the recipe on *Rule Ordering and the `[L]` Flag* earlier in this chapter.

Note

In `httpd 2.2` and earlier, the rewrite log was configured with the separate `RewriteLog` and `RewriteLogLevel` directives. These were removed in `2.4` in favor of the unified `LogLevel` mechanism.

15.6.5 Serving a Fallback Resource When a File Is Missing

Problem: You want to serve a default image, page, or resource when the requested file doesn’t exist. A thread on “Show Alternate Image if Requested Image is Missing” shows a user trying to display a placeholder sketch image when the actual property sketch JPG is missing.

Approach: `mod_rewrite` with `RewriteCond %{REQUEST_FILENAME} !-f` (traditional), or `FallbackResource` directive (modern, preferred)

The mailing list thread reveals the common mistake: mixing `Redirect` (which doesn’t check file existence) with `RewriteCond` conditions. The `FallbackResource` directive, available since `2.2.16`, is often the simplest solution.

FallbackResource (simplest—no `mod_rewrite` needed):

```
# Serve /index.html for any request that doesn't match a real file
FallbackResource /index.html
```

This is ideal for single-page applications (React, Vue, Angular) that handle routing client-side.

`:module:`mod_rewrite` approach` (when you need more control):

```

RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ /index.html [L]

```

This does the same thing but with three lines instead of one. Use it when you need additional conditions (e.g., excluding API paths from the fallback).

Image fallback — serve a placeholder when the requested image doesn't exist:

```

RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule \.(jpg|jpeg|png|gif|webp)$ /images/placeholder.png [L]

```

This checks whether the requested file exists (!-f), and if it's an image extension that's missing, serves a placeholder instead of a 404.

More targeted fallback — only for a specific directory:

```

<Directory "/var/www/html/sketches">
  RewriteEngine On
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule \.jpg$ /sketches/no-sketch-available.jpg [L]
</Directory>

```

Common mistake: Combining Redirect with RewriteCond. The Redirect directive (from mod_alias) does not respect RewriteCond—it fires unconditionally. If you need a conditional redirect, you must use RewriteRule with [R].

15.6.6 Maintenance Mode (503 Service Unavailable)

Problem: You need to put your site into maintenance mode, returning a 503 response to all requests while you work on the backend. A thread on “enforce 503 response using RewriteRule or Redirect due to temporary maintenance” notes that Redirect supports gone (410) but not 503.

Approach: mod_rewrite with [R=503], or ErrorDocument 503 combined with <If>

:module:`mod_rewrite` approach (the classic method):

```

# Return 503 for all requests except the maintenance page itself
ErrorDocument 503 /maintenance.html

RewriteEngine On
# Don't rewrite the maintenance page or its assets
RewriteCond %{REQUEST_URI} !^/maintenance\.html$
RewriteCond %{REQUEST_URI} !^/css/
RewriteCond %{REQUEST_URI} !^/images/
# Allow your own IP through for testing
RewriteCond %{REMOTE_ADDR} !^203\.0\.113\.10$
RewriteRule ^ - [R=503,L]

```

Note

[R=503] requires that ErrorDocument 503 points to a *local* path (not a URL). If you use a URL like `http://...`, httpd sends a 302 redirect to the maintenance page instead of a 503—which search engines interpret as “this page has moved,” not “this site is temporarily down.”

File-existence toggle (enable/disable without editing config):

```
# Enable maintenance mode by creating /var/www/maintenance-flag
# Disable it by removing the file
RewriteEngine On
RewriteCond /var/www/maintenance-flag -f
RewriteCond %{REMOTE_ADDR} !^203\.0\.113\.10$
RewriteCond %{REQUEST_URI} !^/maintenance\.html$
RewriteRule ^ - [R=503,L]

ErrorDocument 503 /maintenance.html
```

This is convenient—deploy maintenance mode with `touch /var/www/maintenance-flag` and disable with `rm /var/www/maintenance-flag`. No config reload needed.

<If> approach (httpd 2.4+, no mod_rewrite needed):

```
<If "-f '/var/www/maintenance-flag' && \
    ! %{REMOTE_ADDR} -ipmatch '203.0.113.10' && \
    ! %{REQUEST_URI} =~ m#^/maintenance\.html$#">
    ErrorDocument 503 /maintenance.html
    Redirect 503 /
</If>
```

Per-application maintenance with RewriteMap (for multi-tenant setups):

```
# Map file: app-status.txt
# app1 down
# app2 up
# app3 down

RewriteMap appstatus "txt:/etc/httpd/conf/app-status.txt"

# /app1/* returns 503 because app1 is "down"
RewriteCond ${appstatus:$1} =down
RewriteRule ^/([^/]+)/ - [R=503,L]

ErrorDocument 503 /maintenance.html
```

Toggling an app into or out of maintenance is a one-line edit to the map file—no restart required, since httpd re-reads text maps automatically.

15.6.7 Handling Special Characters and Encoded URLs

Problem: Rewrite rules break when URLs contain special characters (spaces as %20, international characters, backslashes, etc.). A thread on “404 rewrite error using special character \” shows issues with backslashes, while “chinese char URL encoding/decoding fails” documents problems with multibyte character encoding.

Approach: mod_rewrite with [B] flag and [NE] flag

How httpd handles URL encoding: Before rewrite rules see the URI, httpd decodes percent-encoded characters. So a request for /caf%C3%A9 arrives at the rewrite engine as /café. This is usually helpful, but it creates problems when backreferences are substituted into the target—the decoded characters may need to be re-encoded.

The [B] flag (escape backreferences): tells the rewrite engine to percent-encode special characters in backreferences before substituting them into the target. Without it, spaces, ampersands, and other special characters in captured groups can break the resulting URL.

```
# Without [B]: /search/hello world -> /results?q=hello world (broken URL)
# With [B]:    /search/hello world -> /results?q=hello%20world (correct)
RewriteRule ^/search/(.*)$ /results?q=$1 [B,L]
```

The [NE] flag (no-escape output): prevents the rewrite engine from escaping special characters in the *output*. Useful when your target URL intentionally contains characters like # (fragment identifiers) or % (already-encoded sequences):

```
# Redirect to a URL with a fragment identifier
# Without [NE]: /old#section -> /new%23section (broken)
# With [NE]:    /old#section -> /new#section (correct)
RewriteRule ^/old$ /new#section [NE,R=301,L]

# Preserve already-encoded characters in a redirect
# Without [NE]: %20 gets double-encoded to %2520
RewriteRule ^/(.*)$ https://new.example.com/$1 [NE,R=301,L]
```

Encoded slashes (%2F): By default, httpd rejects URLs containing %2F with a 404, because a percent-encoded slash could be used to bypass <Directory> or <Location> restrictions. If your application legitimately uses %2F in path segments (e.g., Base64-encoded tokens), enable it:

```
# Allow %2F in URLs (use with caution)
AllowEncodedSlashes On

# Or decode them to real slashes (even more caution)
AllowEncodedSlashes NoDecode
```

With NoDecode, the %2F is passed through to the backend without decoding, so the application sees the literal %2F. With On, httpd decodes it to / before the application sees it.

Backslashes: On Windows, httpd converts backslashes to forward slashes automatically. On Unix, a backslash in a URL is technically legal but unusual. If your URLs contain backslashes, use [B] to ensure they’re properly encoded:

```
# Handles paths like /files/path\to\file
RewriteRule ^/files/(.*)$ /download?path=$1 [B,L]
```

Common mistake: Combining [B] and [NE] on the same rule. They have opposite effects—[B] encodes backreferences while [NE] prevents encoding of the output. Use one or the other depending on whether your problem is under-encoding or over-encoding.

15.6.8 Performance with Large Numbers of Redirects

Problem: You have hundreds or thousands of redirects and are concerned about performance impact. The “Redirects and rewrites and performance” thread asks directly: “At what point does it begin to affect performance with the number of redirects?” for a site with ~10,000 accumulated redirects.

Approach: RewriteMap (DBM or text file), database-backed lookups

Individual RewriteRule directives are evaluated sequentially—httpd tests each regex in order until one matches. Ten rules are fine. A hundred is manageable. But at 1,000+, each request walks through a long chain of regex evaluations, and the cost adds up.

Rough guidance:

- **< 100 redirects:** Individual Redirect or RewriteRule directives are fine. No measurable performance impact.
- **100–1,000 redirects:** You may notice a few milliseconds of added latency per request. Consider switching to a map.
- **1,000+ redirects:** Use a RewriteMap. The sequential regex scan becomes the dominant cost of request processing.

Migrating to a RewriteMap:

Convert individual rules to a text map file:

```
# Before: 5,000 individual rules in httpd.conf
RewriteRule ^/old/page-1$ /new/page-1 [R=301,L]
RewriteRule ^/old/page-2$ /new/page-2 [R=301,L]
# ... 4,998 more ...

# After: a single rule with a map lookup
RewriteMap redirects "txt:/etc/httpd/conf/redirect-map.txt"

RewriteCond ${redirects:$1|NOT_FOUND} !NOT_FOUND
RewriteRule ^/(.*)$ ${redirects:$1} [R=301,L]
```

The map file:

```
# redirect-map.txt (one key/value pair per line)
old/page-1    /new/page-1
old/page-2    /new/page-2
```

For maximum performance, convert the text map to DBM format:

```
httxt2dbm -i redirect-map.txt -o redirect-map.db
```

Then reference the DBM map:

```
RewriteMap redirects "dbm:/etc/httpd/conf/redirect-map.db"
```

A DBM lookup is a hash table operation— $O(1)$ regardless of whether the map contains 100 or 100,000 entries. The text map is also $O(1)$ after initial load (httpd reads it into a hash table at startup), but the DBM format loads faster and uses less memory for very large maps.

Updating the map: Edit the text file and run `httxt2dbm` again. For the text map type, httpd detects file changes and reloads automatically. For DBM maps, a graceful restart is needed:

```
httxt2dbm -i redirect-map.txt -o redirect-map.db
apachectl graceful
```

See *Chapter 8* for a detailed treatment of all map types and their performance characteristics.

15.7 When NOT to Use mod_rewrite

As discussed throughout this book, `mod_rewrite` is powerful but often not the best tool for the job. These recipes show problems that are better solved with other modules.

15.7.1 Simple Redirects: Use Redirect, Not RewriteRule

Problem: You're using `RewriteRule` with `[R=301]` for simple page-to-page or site-to-site redirects. A mailing list thread titled “redirect vs. rewrite” directly asks: “What is the difference between `Redirect permanent /` and `RewriteRule ^/(.*) [R,L]?`”

Approach: `Redirect` / `RedirectMatch` from `mod_alias`

The answer from the mailing list: for simple redirects, they're functionally equivalent, but `Redirect` is clearer, faster (no regex engine involved for plain `Redirect`), and less error-prone. As one respondent notes: “Golden rule: if source ends in trailing slash, target must also end in trailing slash.”

Task	<code>Redirect</code>	<code>RedirectMatch</code>	<code>RewriteRule [R]</code>
Single page redirect	Best ✓	Works	Overkill
Redirect preserving path	Best ✓	Works	Overkill
Regex-based redirect	No	Best ✓	Works
Conditional redirect (header, IP, etc.)	No	No	Required ✓
Redirect with query string manipulation	No	No	Required ✓

Examples:

```
# Single page --- Redirect is simplest
Redirect permanent /old-page.html /new-page.html

# Entire directory --- Redirect preserves path automatically
Redirect permanent /blog/ https://blog.example.com/

# Pattern-based --- RedirectMatch when you need regex
RedirectMatch 301 ^/user/([0-9]+)$ /profile/$1

# Conditional --- RewriteRule only when you need RewriteCond
RewriteEngine On
RewriteCond %{HTTP_HOST} ^old\.example\.com$
RewriteRule ^/(.*)$ https://new.example.com/$1 [R=301,L]
```

The golden rule of trailing slashes: when using Redirect to redirect a directory, if the source ends with a trailing slash, the target must also end with a trailing slash. Otherwise, the path appending produces mangled URLs:

```
# CORRECT: both end with /
Redirect permanent /old-section/ /new-section/
# /old-section/page.html -> /new-section/page.html

# WRONG: missing trailing slash on target
Redirect permanent /old-section/ /new-section
# /old-section/page.html -> /new-sectionpage.html (broken!)
```

Redirect has no regex engine overhead. For a plain path-to-path redirect, it's the fastest option. Reaching for RewriteRule when Redirect would do the job is the most common case of mod_rewrite overuse in the wild.

15.7.2 Proxying: Use ProxyPass, Not RewriteRule [P]

Problem: You're using RewriteRule with the [P] flag to proxy requests to a backend server.

Approach: ProxyPass / ProxyPassReverse from mod_proxy

ProxyPass is the right tool for reverse proxying:

```
ProxyPass "/app/" "http://backend.local:8080/app/"
ProxyPassReverse "/app/" "http://backend.local:8080/app/"
```

The equivalent mod_rewrite approach:

```
RewriteEngine On
RewriteRule ^/app/(.*)$ http://backend.local:8080/app/$1 [P]
ProxyPassReverse "/app/" "http://backend.local:8080/app/"
```

Both proxy the request, but ProxyPass is preferred for several reasons:

Feature	ProxyPass	RewriteRule [P]
Connection pooling	Yes (keeps persistent connections to backend)	No (new connection per request)
Load balancing	Yes (BalancerMember)	Manual only
Health checks	Yes (mod_proxy_hcheck)	No
Error handling	ProxyErrorOverride, retry logic	Minimal
Header rewriting	ProxyPassReverse integrated	Still need ProxyPassReverse
WebSocket support	Via mod_proxy_wstunnel	Fragile

When [P] is actually needed: Use it when you need conditional proxying that ProxyPass can't express—for example, proxying only when a specific cookie is present, or routing to different backends based on a regex capture:

```
# Route to different backends based on API version in the URL
RewriteEngine On
RewriteRule ^/api/v1/(.*)$ http://backend-v1:8080/$1 [P]
RewriteRule ^/api/v2/(.*)$ http://backend-v2:8080/$1 [P]
```

Even here, consider ProxyPass with <Location> blocks first. [P] should be your last resort, not your first instinct.

15.7.3 Conditional Logic: Use <If> Expressions

Problem: You're using mod_rewrite for conditional configuration that doesn't involve URL rewriting—like setting headers based on request properties or denying access based on complex conditions.

Approach: <If> expressions (available since httpd 2.4)

The <If> directive supports a rich expression language that can test request headers, environment variables, IP addresses, and more—without the cognitive overhead of RewriteCond/RewriteRule syntax.

Here are common RewriteCond patterns and their <If> equivalents.

Blocking by User-Agent:

```
# mod_rewrite approach
RewriteEngine On
RewriteCond %{HTTP_USER_AGENT} (BadBot|EvilScrapper) [NC]
RewriteRule ^ - [F]

# <If> approach --- clearer intent
<If "%{HTTP_USER_AGENT} =~ /BadBot|EvilScrapper/i">
  Require all denied
</If>
```

Checking a request header:

```
# mod_rewrite: set header based on another header
RewriteCond %{HTTP:X-Forwarded-Proto} =https
RewriteRule ^ - [E=PROTO:https]

# <If> approach
<If "req('X-Forwarded-Proto') == 'https'">
    Header set Strict-Transport-Security "max-age=31536000"
</If>
```

Checking source IP:

```
# mod_rewrite --- awkward regex on IP
RewriteCond %{REMOTE_ADDR} !^10\.
RewriteRule ^/admin - [F]

# <If> --- proper CIDR matching
<If "! %{REMOTE_ADDR} -ipmatch '10.0.0.0/8'">
    <Location "/admin">
        Require all denied
    </Location>
</If>
```

Combining multiple conditions (AND / OR):

```
# mod_rewrite: conditions are implicitly AND
RewriteCond %{REMOTE_ADDR} !^10\.
RewriteCond %{HTTP_USER_AGENT} !InternalMonitor
RewriteRule ^/status - [F]

# <If>: explicit boolean operators
<If "! %{REMOTE_ADDR} -ipmatch '10.0.0.0/8' && \
    %{HTTP_USER_AGENT} !~ /InternalMonitor/">
    <Location "/status">
        Require all denied
    </Location>
</If>
```

The `<If>` approach is preferred when you're not actually rewriting the URL—you're just making access control or header decisions. The expression syntax is documented in the [Apache Expressions](#) reference.

15.7.4 Fallback Resources: Use `FallbackResource`, Not `RewriteRule`

Problem: You want all requests for non-existent files to be handled by a single script (the front controller pattern). The traditional approach uses `mod_rewrite`, but `FallbackResource` does this in a single line.

Approach: `FallbackResource` directive

```
FallbackResource /index.php
```

This single line replaces the entire WordPress-style rewrite block for the common case. It was added in httpd 2.2.16 specifically to address this extremely common use case.

The traditional mod_rewrite front controller block looks like this:

```
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
```

FallbackResource replaces all four lines with one:

```
FallbackResource /index.php
```

Both achieve the same result: if the requested file or directory doesn't exist, serve /index.php instead. The FallbackResource version is faster (no regex evaluation), clearer, and immune to the .htaccess re-processing loop that trips up so many users.

To disable FallbackResource in a subdirectory (e.g., an /admin area that should show real 404s):

```
<Directory "/var/www/html/admin">
  FallbackResource disabled
</Directory>
```

Limitations of FallbackResource:

- Cannot apply conditions (e.g., “only for certain file extensions”)
- Cannot route to different scripts based on URL pattern
- Cannot rewrite the URL—it always serves the fallback as-is
- Does not set PATH_INFO the way mod_rewrite does

If you need conditional routing or multiple front controllers, you still need mod_rewrite. But for the overwhelmingly common case of a single front controller (WordPress, Laravel, Symfony, Drupal), FallbackResource is the right tool.

15.7.5 Advanced: Using RewriteMap for Dynamic Rewrites

Problem: You need dynamic URL mapping that's too complex for static rules—looking up redirects in a database, calling an external program, or using complex logic. Threads on “Perl prg RewriteMap always returns blank” and “RewriteMap prg: How to pass value from Python3 script back to Apache24?” show users struggling with the external program map type.

Approach: RewriteMap with various map types (txt, dbm, prg, dbd, int)

RewriteMap defines a named mapping function that can be called from RewriteRule and RewriteCond substitutions. It must be defined in server config (not `.htaccess`), but the maps it creates can be used anywhere. See *Chapter 8* for the full treatment.

Text file map (``txt``) — simple key/value pairs:

```
# Define the map (server config only)
RewriteMap redirects "txt:/etc/httpd/conf/redirect-map.txt"

# Use it
RewriteEngine On
RewriteCond ${redirects:$1} !=""
RewriteRule ^/(.*)$ ${redirects:$1} [R=301,L]
```

The map file is a plain text file with one key/value pair per line:

```
# redirect-map.txt
old-page.html /new-section/updated-page.html
products/legacy /catalog/current
```

DBM map (``dbm``) — hashed lookup, O(1) performance:

```
RewriteMap redirects "dbm:/etc/httpd/conf/redirect-map.db"
```

Convert a text map to DBM format using `httxt2dbm`:

```
httxt2dbm -i redirect-map.txt -o redirect-map.db
```

This is the right choice for maps with thousands of entries. See the performance recipe later in this chapter.

External program map (``prg``) — call an external script:

```
RewriteMap mymap "prg:/usr/local/bin/rewrite-lookup.py"

RewriteRule ^/user/(.+)$ ${mymap:$1} [L]
```

The program receives lookup keys on stdin (one per line) and must return results on stdout (one per line).

Critical pitfalls:

1. **Buffering:** stdout *must* be line-buffered or unbuffered. In Python, use `print(..., flush=True)` or run with `PYTHONUNBUFFERED=1`. In Perl, set `$| = 1`; . If the program buffers its output, httpd will hang waiting for a response.
2. **Persistence:** the program starts once and runs for the lifetime of the httpd process. It must loop forever reading stdin.
3. **Crashes:** if the program exits, all subsequent lookups return empty strings with no error in the log. Check the error log for startup failures.

A minimal Python example:

```
#!/usr/bin/env python3
import sys
# Must flush every line --- httpd is waiting for the response
for line in sys.stdin:
    key = line.strip()
    # Your lookup logic here
    result = lookup_user(key)
    print(result or "NOT_FOUND", flush=True)
```

Database map (``dbd``) — SQL lookup via `mod_dbd`:

```
# Requires mod_dbd to be configured with a database connection
RewriteMap mymap "dbd:SELECT target FROM redirects WHERE source = %s"

RewriteRule ^/(.*)$ ${mymap:$1} [R=301,L]
```

This queries the database for every request that matches the rule. Use connection pooling (`DBDMin`, `DBDKeep`, `DBDMax`) to manage database load.

Built-in functions (``int``) — string transformations:

```
RewriteMap lowercase "int:tolower"
RewriteRule ^/(.*)$ /${lowercase:$1} [R=301,L]
```

Available functions: `tolower`, `toupper`, `escape`, `unescape`.

15.7.6 Advanced: IP Range Matching with RewriteMap

Problem: You need to match client IPs against CIDR ranges in rewrite rules. A thread on “Apache rewritermap condition that will CIDR-ipmatch against returned value from the map?” shows this is not straightforward with standard rewrite conditions.

Approach: `RewriteMap` with `prg`: type for CIDR matching, or `<If>` with `-ipmatch` operator (preferred for simple cases)

The simplest approach uses the `<If>` directive with the `-ipmatch` operator, available since `httpd 2.4`:

```
# Block access from a specific CIDR range
<If "%{REMOTE_ADDR} -ipmatch '192.168.0.0/16'">
    Require all denied
</If>

# Redirect internal users to an intranet version
<If "%{REMOTE_ADDR} -ipmatch '10.0.0.0/8'">
    RedirectMatch ^/portal(.*)$ https://intranet.example.com$1
</If>
```

You can combine multiple ranges:

```
<If "%{REMOTE_ADDR} -ipmatch '10.0.0.0/8' || %{REMOTE_ADDR} -ipmatch '172.16.0.0/
↪12'">
    # Allow access for both RFC 1918 ranges
    Require all granted
</If>
```

For dynamic CIDR lookups—where the ranges change frequently or are stored externally—use a RewriteMap with an external program:

```
# In server config (not .htaccess)
RewriteMap cidrcheck "prg:/usr/local/bin/cidr-check.py"

RewriteEngine On
RewriteCond ${cidrcheck:%{REMOTE_ADDR}} =blocked
RewriteRule ^ - [F]
```

The external program reads an IP address on stdin and returns blocked or allowed on stdout, checking it against a list of CIDR ranges:

```
#!/usr/bin/env python3
import sys
import ipaddress

BLOCKED_RANGES = [
    ipaddress.ip_network('198.51.100.0/24'),
    ipaddress.ip_network('203.0.113.0/24'),
]

# Line-buffered output is critical
for line in sys.stdin:
    ip = line.strip()
    try:
        addr = ipaddress.ip_address(ip)
        if any(addr in net for net in BLOCKED_RANGES):
            print("blocked", flush=True)
        else:
            print("allowed", flush=True)
    except ValueError:
        print("allowed", flush=True)
```

Recommendation: For static CIDR ranges, `<If> -ipmatch` is dramatically simpler and performs better. Use the RewriteMap approach only when the ranges must be loaded from an external source or change without restarting httpd.

See [Chapter 8](#) for more on external program maps and their buffering requirements.

GLOSSARY

.htaccess

A per-directory configuration file, placed in a directory served by httpd, that applies directives to that directory and its children. Rewrite rules in `.htaccess` behave differently than those in server config — see *RewriteBase* and Chapter 3.

AllowOverride

A directive that controls which types of directives are permitted in `.htaccess` files. `AllowOverride All` enables everything; `AllowOverride None` (the 2.4 default) disables `.htaccess` entirely.

Apache HTTP Server

The web server software, commonly called “httpd.” This book covers version 2.4 and later, with notes on trunk (future 2.5.x) features.

backreference

A captured group from a regular expression match, referenced as `$1` through `$9` in `RewriteRule` substitutions and `%1` through `%9` for `RewriteCond` captures. `$0` and `%0` refer to the entire matched string.

CGI

Common Gateway Interface — a protocol for web servers to execute external programs and return their output as HTTP responses. `ScriptAlias` designates a directory whose contents are treated as CGI programs.

CondPattern

The second argument to `RewriteCond` — the pattern or expression that the `TestString` is compared against.

content negotiation

The process by which httpd selects the best representation of a resource based on the client’s `Accept-*` headers. `mod_negotiation` and `MultiViews` implement this. See Chapter 2.

directive

A configuration command in the httpd configuration file (or `.htaccess`). Examples: `RewriteRule`, `DocumentRoot`, `ProxyPass`.

DocumentRoot

The directory from which httpd serves static files by default. A request for `/page.html` maps to `DocumentRoot/page.html`.

ErrorDocument

A directive that specifies a custom response for a given HTTP status code. `ErrorDocument 404 /not-found.html` serves a custom 404 page.

FallbackResource

A `mod_dir` directive that specifies a default handler for requests that don't match any existing file — the mechanism behind most front-controller frameworks. See Chapter 2.

flag

A modifier enclosed in square brackets at the end of a `RewriteRule`, such as `[L]`, `[R=301]`, or `[P]`. Flags alter how the rule is processed. See Chapter 6.

front controller

A web application design pattern where all requests are routed through a single entry point (typically `index.php` or `app.py`). `FallbackResource` and `mod_rewrite` both support this pattern.

handler

An internal `httpd` representation of the action to be taken when a file is called. The `[H]` flag (or `SetHandler`) assigns a handler to matched requests.

httpd

The Apache HTTP Server daemon. The executable is typically called `httpd` or `apache2` depending on the distribution.

MIME type

A label identifying the type of content, such as `text/html` or `image/png`. `httpd` uses MIME types (via `mod_mime`) to determine how to serve files.

mod_alias

The module providing `Alias`, `AliasMatch`, `Redirect`, `RedirectMatch`, `ScriptAlias`, and `ScriptAliasMatch` directives. Simpler and faster than `mod_rewrite` for static URL mapping.

mod_proxy

The core proxy module. By itself it provides the framework (`ProxyPass`, `ProxyPassReverse`); pair it with a protocol module like `mod_proxy_http` or `mod_proxy_fcgi` for actual proxying. See Chapter 9.

mod_rewrite

The rule-based URL rewriting engine. Uses PCRE regular expressions to match and transform request URLs. The subject of most of this book.

MultiViews

An `Options` setting that enables content negotiation via `mod_negotiation`. A request for `/doc` will match `doc.en.html`, `doc.fr.html`, etc., based on the client's language preferences.

PCRE

Perl Compatible Regular Expressions — the `regex` library used by `mod_rewrite`. Syntax is documented in `man pcre2pattern` or `man perlre`.

per-directory context

Configuration that applies within a `<Directory>`, `<DirectoryMatch>`, `<Files>`, or `<FilesMatch>` block, or in a `.htaccess` file. All of these are per-directory context — `.htaccess` is not a separate context type but simply another way to specify per-directory configuration. Rewrite rules in this context operate on the URL-path with the directory prefix stripped. See `RewriteBase` and Chapter 3.

ProxyPass

The primary `mod_proxy` directive for reverse proxying. Maps a local URL prefix to a backend server URL.

ProxyPassReverse

Rewrites `Location`, `Content-Location`, and `URI` headers in the backend's response so redirects point to the proxy's URL rather than the backend's internal URL.

regex

Regular expression — a pattern-matching language. In the context of this book, PCRE regex as used by `RewriteRule` and `RewriteCond`. See Chapter 1.

reverse proxy

A server that accepts client requests and forwards them to one or more backend servers, returning the response as if it originated from the proxy itself. `mod_proxy` with `ProxyPass` is the standard `httpd` reverse proxy. The `[P]` flag in `RewriteRule` also triggers proxying.

RewriteBase

A directive that sets the base URL for per-directory rewrites. Only meaningful in `.htaccess` or `<Directory>` context. See Chapter 3.

RewriteCond

A directive that attaches a condition to the following `RewriteRule`. The rule fires only if all preceding conditions match. See Chapter 7.

RewriteEngine

Enables or disables the rewriting engine. Must be set to `On` in each context (server, virtual host, directory) where you want rules to be processed.

RewriteMap

A directive that defines a named mapping function (text file, DBM, program, database query, or internal function) for use in `RewriteRule` and `RewriteCond` substitutions. See Chapter 8.

RewriteOptions

Controls special behaviors of the rewrite engine — rule inheritance, URI handling, and other edge cases. See Chapter 3.

RewriteRule

The central directive of `mod_rewrite`. Matches a URL pattern and transforms it into a new URL or triggers an action (redirect, proxy, forbid, etc.). See Chapter 4.

server context

Configuration that applies at the top level of `httpd.conf` or within a `<VirtualHost>` block, as opposed to per-directory context.

server variable

A value available for testing in `RewriteCond` via the `%{VARIABLE_NAME}` syntax. Examples: `%{HTTP_HOST}`, `%{REQUEST_URI}`, `%{REMOTE_ADDR}`. See Chapter 7.

substitution

The second argument to `RewriteRule` — the replacement URL or path. May contain backreferences (`$1`), server variables, and `RewriteMap` expansions.

TestString

The first argument to `RewriteCond` — the string to test, which can contain server variables, backreferences, and map expansions.

URL mapping

The process by which `httpd` determines what resource corresponds to a requested URL. Involves `DocumentRoot`, `Alias`, `Redirect`, `mod_rewrite`, and other modules. See Chapter 2.

URL-path

The path component of a URL, without the scheme, host, or query string. For `http://example.com/one/two?q=x`, the URL-path is `/one/two`.

virtual host

A configuration block (`<VirtualHost>`) that allows a single `httpd` instance to serve multiple websites. Virtual hosts can be name-based (distinguished by `Host:` header) or IP-based. See Chapter 10.

LIST OF EPIGRAPHS

Each chapter opens with an epigraph drawn from a different author. This page collects them in one place, with notes on why each was chosen.

17.1 Preface

They stared at the branch. There wasn't just one flower out there, there were dozens, although the frogs weren't able to think like this because frogs can't count beyond one.

They saw lots of ones.

—Terry Pratchett, *Wings*

Notes:

The frogs see “lots of ones” — individual problems without the bigger picture. The book teaches you to count higher: to see the patterns behind the individual redirects and rewrite rules.

Wings is the third book in the Nome Trilogy, also known as the Bromeliad, by Terry Pratchett. It tells the story of traveling far away to find home, and is one of our (Maria and me) favorite books. You should read it. It's way better than this book. -rcb

17.2 Chapter 1: Regular Expressions

The first thing you learn in life is you're a fool.

The last thing you learn in life is you're the same fool.

—Ray Bradbury, *Dandelion Wine*

Notes:

Regular expressions are the first thing the book teaches, and the first place every reader feels foolish. You'll return to them throughout your career and feel the same way again.

Every summer I read *Dandelion Wine*. It's a bittersweet story about growing up, and learning, and losing, and I find something new in it every time. Hope you have the same experience with this book. -rcb

17.3 Chapter 2: URL Mapping

Some people are heroes. And some people jot down notes.
Sometimes they're the same person.

—Terry Pratchett, *The Truth*

Notes:

Not every contribution is heroic. Some people write the documentation, maintain the maps, keep the config files clean. All skills are valuable.

I got involved in open source because I'm a writer, not a programmer. I stayed involved because that was celebrated, and because the communities can be delightful. I'm still not a programmer, although I tried to be for a few years. But this quote always reminds me that all skills are valuable. -rcb

17.4 Chapter 3: Introduction to mod_rewrite

In the high and far-off times the Elephant, O Best Beloved, had no trunk.

—Rudyard Kipling, *The Elephant's Child*

Notes:

Before this chapter, you had no trunk — no mod_rewrite. The Elephant's Child got its trunk by sticking its nose where it didn't belong and having it stretched by a crocodile. Most people learn mod_rewrite the same way.

I used to read the *Just So Stories* to the kids when they were little, and I did all the voices. *The Elephant's Child* was always a favorite. 'Led go! You are hurtig be!' Consider doing the voices while reading this chapter. -rcb

17.5 Chapter 4: RewriteRule

Quickly, bring me a beaker of wine, that I may wet
my brain and say something clever.

—Aristophanes (attributed)

Notes:

The moment you sit down to write your first RewriteRule and need some liquid courage to say something clever. Perhaps apocryphal, but whatevs. -rcb

17.6 Chapter 5: Rewrite Logging

Oh, I have slipped the surly bonds of earth
And danced the skies on laughter-silvered wings.

—John Gillespie Magee Jr., *High Flight*

Notes:

Rising above the chaos to see clearly — which is exactly what the rewrite log gives you.

This poem celebrates finding the beautiful amidst the ugly. It was written by a fighter pilot who died in a mid-air collision over England in 1941, at the age of 19. I learned it in high school and it's stayed with me ever since. -rcb

17.7 Chapter 6: RewriteRule Flags

My girl came to the study
and said Help me;
I told her I had a time problem
which meant:
I would die for you but I don't have ten minutes.

—Brenda Hillman, *Time Problem*

Notes:

Every flag is a choice about how to spend your processing time. [L] says stop here. [S] skips ahead. [END] says we're done.

This poem always reminds me that our most valuable asset is our time, and we choose how to spend it. It's always a choice, so choose wisely, and prioritize what actually matters. -rcb

17.8 Chapter 7: RewriteCond

Snore on in your front row seat
Let not my voice disturb the wordless heaven your eyes have found

—James Kirkup, *To An Old Lady Asleep At A Poetry Reading*

Notes:

RewriteCond checks whether anyone is paying attention before the rule fires. Sometimes the answer is no, and that's fine — the request passes through untouched.

I won the Kenya national poetry competition reciting this one, and I think of it every time I'm on stage and see someone in the audience who isn't quite paying attention. -rcb

17.9 Chapter 8: RewriteMap

And memories, he knew, were not glass treasures to be
kept locked within a box. They were bright ribbons to
be hung in the wind.

—Terry Brooks, *The Talismans of Shannara*

Notes:

RewriteMaps externalize knowledge — lookups, translations, redirects — into files and programs rather than locking them inside rewrite rules.

The Shannara books, by Terry Brooks, were what got me into reading high fantasy, and warrant re-reading regularly, although there are 33 of them, so that’s a pretty serious undertaking. -rcb

17.10 Chapter 9: Proxies and mod_rewrite

“His voice,” thought Will, “I never noticed. It’s the same color as his hair.”

—Ray Bradbury, *Something Wicked This Way Comes*

Notes:

A proxy *is* the voice that speaks for the backend — and if it’s doing its job well, you never notice it’s not the real thing. Bradbury’s synesthesia captures that perfectly: the voice and the appearance blur together into something you accept without question.

If you haven’t read *Something Wicked*, you should. It’s best if you read it as a kid, and then again as an adult, but we can’t have everything. Did I mention how much I love Bradbury? -rcb

17.11 Chapter 10: Virtual hosts and mod_rewrite

When you’ve only got two ducks, they’re always in a row.

—Rich Bowen

Notes:

If you’ve only got one virtual host, you don’t need virtual hosts. The complexity of mass virtual hosting only matters once you have enough ducks to worry about their arrangement.

Also, I’m a Jeep driver. You’ve seen those Jeeps with way too many ducks on their dashboards, right? Guess how many I have. -rcb

17.12 Chapter 11: Access control with mod_rewrite

When the dragons grow too mighty
To slay with pen or sword
I grow weary of the battle
And the storm I walk toward
When all around is madness

And there's no safe port in view
I long to turn my path homeward
To stop awhile with you

—Rush, *Madrigal*

Notes:

Keeping the dragons out, and knowing when to stop fighting. The whole chapter is about access control, and the closing section is about when to put the sword down and use simpler tools instead.

And of course we have to have a Rush quote. If you're a Rush fan you know that it's practically impossible to choose just one. (Rest well, Neil) This is one of the most beautiful of Rush's songs, and not much known outside of the fans. It expresses the beauty of having a true friend to rest with from the struggles of the world.
-rcb

17.13 Chapter 12: Conditional Configuration

Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke, *Profiles of the Future*

Notes:

The chapter where the config file stops being static and starts being programmable. To someone used to plain directives, `<If>` expressions and `Define` variables look like magic.

I met Clarke in Colombo, Sri Lanka, at ApacheCon Asia 2006. All I could think of to say to him was “Thank you for the stories.” -rcb

17.14 Chapter 13: Content Munging

On the Coast of Coromandel
Where the early pumpkins blow,
In the middle of the woods
Lived the Yonghy-Bonghy-Bo.

—Edward Lear, *The Courtship of the Yonghy-Bonghy-Bo*

Notes:

Whimsical nonsense for a chapter about transforming content into something new — munging things into strange and wonderful shapes.

Lear's nonsense was another staple for reading to the kids. The Jumblies and the Pobble were also favorites. Lovely brain candy after too much seriousness in the world. -rcb

17.15 Chapter 14: Recipes

Not to know that no space of regret can make amends
for one life's opportunity misused!

—Charles Dickens, *A Christmas Carol*

Notes:

Jacob Marley's ghost, lamenting wasted opportunity — a fitting epigraph for a chapter full of recipes to avoid common mistakes.

A Christmas Carol is another favorite book, and I read it every year too, and watch several of the movies. Patrick Stewart's version is, of course, the best, closely followed by the Muppet one. -rcb

Symbols

- ! (*NOT*)
 - RewriteRule, 40
- (), 9
- () (*grouping/capturing*)
 - regular expressions, 9
- *, 7
- * (*zero or more*)
 - regular expressions, 7
- +, 7
- + (*one or more*)
 - regular expressions, 7
- ., 5
- . (*dot*)
 - regular expressions, 5
- .htaccess, 1, 32, **171**
 - configuration, 1
 - limitations, 35
 - overview, 32
 - path stripping, 34
 - perdir prefix stripping, 51
 - restrictions, 34
 - rewrite log example, 52
 - rewrite logging, 51
 - RewriteRule, 43
- .htaccess context
 - recipes, 154
- .htaccess files
 - mod_rewrite, 32
- ?, 8
- ? (*optional/non-greedy*)
 - regular expressions, 8
- \$, 6
- \$ (*dollar sign*)
 - regular expressions, 6
- \$1, 9
 - RewriteRule, 41
- \$N (*RewriteRule*)
 - backreferences, 71
- %1, 9
 - RewriteCond, 41
- %N (*RewriteCond*)
 - backreferences, 71
- %{VARNAME}
 - RewriteRule, 42
- (*dash*)
 - RewriteRule target, 41
- D flag
 - command-line switches, 112
- F (*subrequest file*)
 - RewriteCond, 74
- U (*subrequest URL*)
 - RewriteCond, 74
- d (*directory test*)
 - RewriteCond, 80
- d (*is directory*)
 - RewriteCond, 74
- f (*file test*)
 - RewriteCond, 80
- f (*is regular file*)
 - RewriteCond, 74
- l (*is symlink*)
 - RewriteCond, 74
- s (*has size*)
 - RewriteCond, 74
- x (*executable*)
 - RewriteCond, 74
- {n,m}, 7
- {n,m} (*repetition*)
 - regular expressions, 7
- ^, 6
- ^ (*caret*)
 - regular expressions, 6
- ~user

- RewriteRule, 44
- ~user URLs, 26
- <If>
 - expression syntax, 115
- [\], 10
- [] (*character class*)
 - regular expressions, 10
- 301 Moved Permanently
 - HTTP status codes, 21
- 301 caching
 - debugging, 53
- 301 redirect, 21
 - recipes, 134
- 302 redirect, 21
 - HTTP status codes, 66
- 403 Forbidden
 - HTTP status codes, 60
- 410 Gone
 - HTTP status codes, 60
- 503 response
 - recipes, 159
- A**
- absolute URL
 - RewriteRule target, 41
- access control
 - environment variables, 108
 - HTTPS redirect, 107
 - image hotlinking, 103
 - IP address blocking, 105
 - referrer-based redirect, 106
 - rewrite/avoid, 103
 - time-based, 106
 - user agent blocking, 104
 - when not to use mod_rewrite, 108
- Action, 24
 - directives, 24
- AddOutputFilterByType, 126
- AH00124, 53, 61
- Alias, 20, 64
 - directives, 20
- Alias interaction
 - virtual hosts, 99
- AliasMatch, 20
 - mod_alias, 20
- AllowAnyURI
 - RewriteOptions, 35
- AllowEncodedSlashes, 55
 - directives, 55
- AllowNoSlash
 - RewriteOptions, 35
- AllowOverride, 32, **171**
 - directives, 32
 - recipes, 154
- Anchor examples, 6
- Anchor, 6
- ap_expr, 73, 114, 115
 - expressions, 73, 115
- Apache HTTP Server, **171**
 - filters, 126
 - mod_rewrite, 1
 - version 2.2, 2
 - version 2.4, 2
 - version 2.4 configuration, 111
- authentication redirect
 - recipes, 143
- B**
- B flag, 55
 - RewriteRule flags, 55
- backreference, **171**
- Backreferences, 9
- backreferences, 39, 41, 55, 71
 - \$N (*RewriteRule*), 71
 - %N (*RewriteCond*), 71
 - RewriteRule, 41
- backrefnoplus
 - RewriteRule flags, 56
- Backslash, 5
- basic [P] example
 - proxy, 94
- BCTLS flag, 56
 - RewriteRule flags, 56
- block by IP
 - recipes, 144
- block by referrer
 - recipes, 141
- block by user agent
 - recipes, 142
- BNE flag, 56
 - RewriteRule flags, 56
- BNP flag, 56
 - RewriteRule flags, 56
- book formats
 - reStructuredText, 2
 - Sphinx, 2

- bot blocking
 - recipes, 142
- C**
- C flag, 57
 - RewriteRule flags, 57
- canonical hostname
 - recipes, 130
- canonical hostname example
 - If directive, 114
- Capturing, 9
- CGI, **171**
 - virtual hosts, 99
- chain
 - RewriteRule flags, 57
- CheckCaseOnly, 26
- CheckSpelling, 26
- clean URLs
 - recipes, 139
- CO flag, 57
 - RewriteRule flags, 57
- combining with OR
 - RewriteCond, 80
- command-line switches
 - D flag, 112
- common mistakes
 - proxy, 96
- conditional
 - configuration, 109
 - logging, 117
 - proxy, 94
- conditional configuration, 109
- conditional fallback
 - recipes, 158
- CondPattern, **171**
 - RewriteCond, 74
- configuration
 - .htaccess, 1
 - conditional, 109
 - server context, 32
 - virtual hosts, 23
- content
 - modification, 120
- content munging, 120
- content negotiation, 1, 25, **171**
- cookie
 - RewriteRule flags, 57
- cookie-based access
 - recipes, 143
- cookies, 57
 - domain, 57
 - HttpOnly flag, 58
 - lifetime, 57
 - path, 57
 - secure flag, 58
- creating
 - RewriteMap, 83
- D**
- dbd (*SQL query*)
 - RewriteMap types, 88
- dbm (*hash file*)
 - RewriteMap types, 87
- DBM hash file, 87
- debugging
 - 301 caching, 53
 - infinite loops, 53
 - mod_rewrite, 46
 - per-directory, 48
 - RewriteMap, 53
- debugging rewrite
 - recipes, 157
- Define, 113
 - directives, 113
- deny list
 - RewriteMap, 105
- directive, **171**
- directives
 - Action, 24
 - Alias, 20
 - AllowEncodedSlashes, 55
 - AllowOverride, 32
 - Define, 113
 - DirectoryIndex, 18
 - DocumentRoot, 17
 - Else, 114
 - ElseIf, 114
 - ErrorDocument, 27
 - ErrorLog, 49
 - FallbackResource, 27
 - FilesMatch, 111
 - If, 114
 - IfDefine, 112
 - IfModule, 111
 - IfVersion, 111
 - IndexOptions, 19

- LoadModule, 29
- Location, 21
- LocationMatch, 21
- LogLevel, 48
- MultiViews, 25
- Options, 19
- Options FollowSymLinks, 34
- ProxyPass, 64
- ProxyPassMatch, 64
- Redirect, 21
- RedirectMatch, 111
- RewriteBase, 37
- RewriteCond, 69
- RewriteEngine, 31
- RewriteMap, 81
- RewriteOptions, 35
- RewriteRule, 38
- ScriptAlias, 20
- SetHandler, 21
- Directory, 14
- Directory block
 - LogLevel, 48
- directory listings, 19
- DirectoryIndex, 18
 - directives, 18
- DirectorySlash
 - recipes, 132
- discardpath
 - RewriteRule flags, 58
- DocumentRoot, 17, **171**
 - directives, 17
- domain
 - cookies, 57
- domain migration
 - recipes, 133
- DPI flag, 58
 - RewriteRule flags, 58
- dynamic
 - virtual hosts, 97
- dynamic backend selection
 - RewriteMap, 95
- E**
- E (*environment variable*)
 - flags, 108
- E flag, 58
 - RewriteRule flags, 58
- E flag; REDIRECT_ prefix
 - RewriteRule flags, 59
- Else
 - directives, 114
- Else directive, 67, 114
- ElseIf
 - directives, 114
- ElseIf directive, 67, 114
- Email address, 11
- END flag, 60
 - loop prevention, 60
 - per-directory context, 60
 - RewriteRule flags, 60
- ENV
 - server variables, 74
- env
 - RewriteRule flags, 58
- environment variables, 58
 - access control, 108
 - logging, 117
 - REDIRECT_ prefix, 59
 - RewriteCond, 108
- ErrorDocument, 27, **172**
 - directives, 27
 - recipes, 166
- ErrorLog, 49
 - directives, 49
- errors
 - Internal Server Error, 31
- Escape, 5
- escape, 85
 - RewriteMap internal functions, 85
- escape backreferences
 - RewriteRule flags, 55
- Examples, 11
- examples
 - greedy matching, 8
 - rewrite logging, 49
 - RewriteCond, 79
- Examples of greedy matching, 8
- expansion order
 - RewriteRule, 42
- expression parser, 114, 115
- expression syntax
 - <If>, 115
- expressions
 - ap_expr, 73, 115
- extensionless URLs
 - recipes, 136

external program map
 recipes, 167

ExtFilterDefine, 127

F

F flag, 60

RewriteRule flags, 60

FallbackResource, 18, 27, 172

directives, 27

mod_dir, 18

recipes, 166

fastdbd

RewriteMap types, 88

file attribute tests

RewriteCond, 74

file existence check

RewriteCond, 80

file not found fallback

recipes, 158

file-system path, 17, 40

RewriteRule target, 40

FilesMatch, 111

directives, 111

FilterChain, 126

FilterDeclare, 126

FilterProvider, 126

filters, 126

Apache HTTP Server, 126

fixup phase

request processing, 29

flag, 172

FLAGS

RewriteRule, 39

flags

E (*environment variable*), 108

P (*proxy*), 93

RewriteCond, 79

RewriteRule, 54

flags summary

RewriteRule, 44

forbidden

RewriteRule flags, 60

force SSL

recipes, 129

Friedl, 3

front controller, 18, 172

recipes, 138

G

G flag, 60

RewriteRule flags, 60

gone

RewriteRule flags, 60

Greedy matching, 8

greedy matching

examples, 8

grep, 16

Grouping, 9

H

H flag, 61

RewriteRule flags, 61

handler, 172

RewriteRule flags, 61

home directory

RewriteRule, 44

hooks

mod_rewrite, 29

hostname matching

RewriteCond, 79

hotlink protection

recipes, 141

hotlinking, 103

hotlinking example

If directive, 114

HTTP (*header lookup*)

server variables, 74

HTTP status codes

301 Moved Permanently, 21

302 redirect, 66

403 Forbidden, 60

410 Gone, 60

HTTP to HTTPS redirect

recipes, 129

HTTP_COOKIE

server variables, 72

HTTP_HOST

server variables, 72

HTTP_REFERER

RewriteCond, 103

server variables, 72

HTTP_USER_AGENT

RewriteCond, 104

server variables, 72

httpd, 172

httpd 2.4

- logging, 48
- httpddbm, 87
- HttpOnly flag
 - cookies, 58
- HTTPS
 - RewriteCond, 107
 - server variables, 72
- HTTPS detection
 - RewriteCond, 80
- HTTPS redirect
 - access control, 107
- I
- If
 - directives, 114
- If directive, 1, 67, 114
 - canonical hostname example, 114
 - hotlinking example, 114
- If expression
 - recipes, 165
- IfDefine, 112
 - directives, 112
- IfModule, 111
 - directives, 111
- IfVersion, 111
 - directives, 111
- IgnoreContextInfo
 - RewriteOptions, 35
- IgnoreInherit
 - RewriteOptions, 35
- image hotlinking
 - access control, 103
- IndexOptions
 - directives, 19
- infinite loop
 - recipes, 152
- infinite loops
 - debugging, 53
- Inherit
 - RewriteOptions, 35
- InheritBefore
 - RewriteOptions, 35
- InheritDown
 - RewriteOptions, 35
- InheritDownBefore
 - RewriteOptions, 35
- input validation
 - security, 45
- InputSed, 123
- int (*internal function*)
 - RewriteMap types, 84
- int:tolower
 - RewriteMap, 97
- interaction with mod_alias
 - mod_rewrite, 32
- internal redirect
 - REDIRECT_ prefix, 59
- Internal Server Error, 31
 - errors, 31
- introduction
 - mod_rewrite, 27
 - URL mapping, 17
- Introduction to regular expressions, 3
- IP address blocking
 - access control, 105
- IP range matching
 - recipes, 169
- IP-based
 - virtual hosts, 23
- IP-based access control
 - recipes, 144
- J
- Jeffrey, 3
- L
- L flag, 61
 - looping, 61
 - per-directory context, 61
 - RewriteRule flags, 61
- LA-F (*filename look-ahead*)
 - server variables, 74
- LA-U (*URL look-ahead*)
 - server variables, 74
- last
 - RewriteRule flags, 61
- LegacyPrefixDocRoot
 - RewriteOptions, 35
- lexicographic comparison, 74
- lifetime
 - cookies, 57
- limitations
 - .htaccess, 35
- LimitInternalRecursion, 53
- loading
 - mod_rewrite, 29

- LoadModule, 29
 - directives, 29
- Location
 - directives, 21
- Location block
 - LogLevel, 48
- LocationMatch
 - directives, 21
- log entry format
 - rewrite logging, 50
- LogFormat
 - per-vhost, 100
- logging
 - conditional, 117
 - environment variables, 117
 - httpd 2.4, 48
 - mod_rewrite, 46
 - per-directory, 119
 - per-directory context, 51
 - per-module, 119
 - pipelined, 120
 - response code conditional, 119
 - RewriteMap, 53
 - virtual hosts, 100
- LogLevel, 48
 - directives, 48
 - Directory block, 48
 - Location block, 48
 - per-directory, 48
 - trace levels, 47
- LongURLOptimization
 - RewriteOptions, 35
- loop detection
 - rewrite logging, 53
- loop prevention
 - END flag, 60
- looping
 - L flag, 61
 - per-directory context, 60
- M**
- maintenance mode
 - recipes, 159
- map lookups
 - rewrite logging, 53
- MapName
 - RewriteMap, 83
- MapSource
 - RewriteMap, 83
- MapType
 - RewriteMap, 83
- mass hosting
 - virtual hosts, 23, 97
- mass reverse proxy
 - mod_proxy, 24
- Mastering Regular Expressions by Jeffrey Friedl, 3
- MergeBase
 - RewriteOptions, 35
- Metacharacter table, 5
- Metacharacters, 5
- MIME type, 67, 172
- mod_actions, 24
 - modules, 24
- mod_alias, 1, 172
 - AliasMatch, 20
 - modules, 20
 - processing order, 32
 - recipes, 163
 - Redirect, 21
 - RedirectMatch, 21
 - ScriptAliasMatch, 20
- mod_autoindex, 19
 - modules, 19
- mod_dir
 - FallbackResource, 18
- mod_ext_filter, 127
- mod_filter, 126
- mod_imagemap, 25
 - modules, 25
- mod_macro, 115
 - modules, 115
- mod_negotiation, 25
 - modules, 25
- mod_proxy, 1, 64, 91, 172
 - mass reverse proxy, 24
 - modules, 24
 - processing order, 32
 - recipes, 164
- mod_proxy_ajp, 91
- mod_proxy_balancer, 91
- mod_proxy_connect, 91
- mod_proxy_express, 24, 91, 116
 - modules, 116
- mod_proxy_fcgi, 91
- mod_proxy_fdpass, 91

- mod_proxy_ftp, 91
- mod_proxy_hcheck, 91
- mod_proxy_html, 124
 - modules, 124
- mod_proxy_http, 91
- mod_proxy_http2, 91
- mod_proxy_scgi, 91
- mod_proxy_uwsgi, 91
- mod_proxy_wstunnel, 91
- mod_rewrite, 1, 17, 27, **172**
 - .htaccess files, 32
 - Apache HTTP Server, 1
 - debugging, 46
 - hooks, 29
 - interaction with mod_alias, 32
 - introduction, 27
 - loading, 29
 - logging, 46
 - modules, 17, 27
 - performance, 64
 - performance impact of logging, 54
 - processing order, 32
 - request processing phases, 29
 - security considerations, 64, 68
 - trace level, 48
 - trace levels, 47
- mod_rewrite alternative
 - mod_userdir, 100
- mod_rewrite alternatives
 - recipes, 165
- mod_rewrite recipe
 - virtual hosts, 97
- mod_sed, 123
 - modules, 123
- mod_speling, 26
 - URL correction, 26
- mod_substitute, 121
 - modules, 121
- mod_userdir, 26
 - mod_rewrite alternative, 100
 - user directories, 26
- mod_vhost_alias, 23, 116
 - modules, 116
 - VirtualDocumentRoot, 23
- mod_vhost_alias comparison
 - virtual hosts, 99
- modification
 - content, 120

- modules
 - mod_actions, 24
 - mod_alias, 20
 - mod_autoindex, 19
 - mod_imagemap, 25
 - mod_macro, 115
 - mod_negotiation, 25
 - mod_proxy, 24
 - mod_proxy_express, 116
 - mod_proxy_html, 124
 - mod_rewrite, 17, 27
 - mod_sed, 123
 - mod_substitute, 121
 - mod_vhost_alias, 116
 - processing order, 32
- moved page redirect
 - recipes, 134
- MultiViews, 25, **172**
 - directives, 25

N

- N flag, 63
 - RewriteRule flags, 63
- name-based
 - virtual hosts, 23
- NC (*nocase*)
 - RewriteCond flags, 79
- NC flag, 63
 - RewriteRule flags, 63
- NE flag, 63
 - RewriteRule flags, 63
- Negation, 11
- negation
 - RewriteRule, 40
- next
 - RewriteRule flags, 63
- nocase
 - RewriteRule flags, 63
- noescape
 - RewriteRule flags, 63
- nosubreq
 - RewriteRule flags, 63
- NS flag, 63
 - RewriteRule flags, 63
- NV (*novary*)
 - RewriteCond flags, 79

O

old domain to new domain
 recipes, 133

open redirect, 45

Optional matching, 8

Options

directives, 19

Options FollowSymLinks

directives, 34

OR (*ornext*)

RewriteCond flags, 79

output filter chain, 126

output filters, 120

OutputSed, 123

overview

.htaccess, 32

P

P (*proxy*)

flags, 93

P flag, 64

RewriteRule flags, 64

pass-through, 41

passthrough

RewriteRule flags, 64

path

cookies, 57

path stripping

.htaccess, 34

path to query string

recipes, 139

path traversal, 45

PATH_INFO, 58

path-based routing

recipes, 138

PATTERN

RewriteRule, 39

pattern matching

RewriteRule, 39

PCRE, 39, 172

pcre2test, 15

per-directory context, 172

per-directory

debugging, 48

logging, 119

LogLevel, 48

per-directory context, 32, 39

END flag, 60

L flag, 61

logging, 51

looping, 60

re-entry, 60

RewriteRule, 43

per-directory rewrite

recipes, 154

per-module

logging, 119

per-vhost

LogFormat, 100

perdir prefix stripping, 52

.htaccess, 51

performance

mod_rewrite, 64

recipes, 162

rewrite logging, 54

performance impact of logging

mod_rewrite, 54

Phone number, 12

piped

logging, 120

prefix stripping

RewriteRule, 43

prg (*external program*)

RewriteMap types, 87

process id

rewrite logging, 50

processing order

mod_alias, 32

mod_proxy, 32

mod_rewrite, 32

modules, 32

RewriteRule, 44

protocol modules

proxy, 91

proxy, 24, 64

basic [P] example, 94

common mistakes, 96

conditional, 94

protocol modules, 91

ProxyPass vs [P], 93

RewriteMap, 95

RewriteRule flags, 64

SSL/TLS, 95

troubleshooting, 96

proxy decisions

RewriteCond, 94

ProxyExpressEnable, 24

ProxyHTMLEnable, 124

ProxyHTMLExtended, 124

ProxyHTMLURLMap, 124

ProxyPass, 64, **173**

directives, 64

ProxyPass and rewrite

recipes, 145

ProxyPass vs mod_rewrite proxy

recipes, 164

ProxyPass vs [P]

proxy, 93

ProxyPassMatch, 64

directives, 64

ProxyPassReverse, 94, **173**

ProxyPreserveHost, 95

PT (*passthrough*)

RewriteRule flags, 40

PT flag, 64

RewriteRule flags, 32, 64

Q

QSA

RewriteRule flags, 42

QSA flag, 65

recipes, 149

RewriteRule flags, 65

qsappend

RewriteRule flags, 65

QSD

RewriteRule flags, 42

QSD flag, 65

RewriteRule flags, 65

qsdiscard

RewriteRule flags, 65

QSL flag, 65

RewriteRule flags, 65

qslast

RewriteRule flags, 65

query string, 39, 42, 65

RewriteRule, 39

query string manipulation

RewriteRule, 42

query string matching

RewriteCond, 79

query string rewrite

recipes, 149

QUERY_STRING

server variables, 72

R

R (*redirect*)

RewriteRule flags, 41

R flag, 66

RewriteRule flags, 66

re-entry

per-directory context, 60

recipes

.htaccess context, 154

301 redirect, 134

503 response, 159

AllowOverride, 154

authentication redirect, 143

block by IP, 144

block by referrer, 141

block by user agent, 142

bot blocking, 142

canonical hostname, 130

clean URLs, 139

conditional fallback, 158

cookie-based access, 143

debugging rewrite, 157

DirectorySlash, 132

domain migration, 133

ErrorDocument, 166

extensionless URLs, 136

external program map, 167

FallbackResource, 166

file not found fallback, 158

force SSL, 129

front controller, 138

hotlink protection, 141

HTTP to HTTPS redirect, 129

If expression, 165

infinite loop, 152

IP range matching, 169

IP-based access control, 144

maintenance mode, 159

mod_alias, 163

mod_proxy, 164

mod_rewrite alternatives, 165

moved page redirect, 134

old domain to new domain, 133

path to query string, 139

path-based routing, 138

per-directory rewrite, 154

- performance, 162
- ProxyPass and rewrite, 145
- ProxyPass vs mod_rewrite proxy, 164
- QSA flag, 149
- query string rewrite, 149
- Redirect vs RewriteRule, 163
- remove file extension, 136
- remove query string, 150
- reverse proxy rewrite, 145
- rewrite log, 157
- rewrite loop, 152
- rewrite rule ordering, 155
- RewriteLog, 157
- RewriteMap, 167
- RewriteMap CIDR, 169
- rule order, 155
- SetEnvIf query string, 151
- special characters in URLs, 160
- strip query string, 150
- thousands of redirects, 162
- TLS termination proxy, 146
- too many redirects, 152
- trailing slash redirect, 132
- URL encoding, 160
- WebSocket proxy, 147
- wildcard subdomain redirect, 135
- wss proxy, 147
- www canonicalization, 130
- X-Forwarded-Proto, 146
- Redirect, 21
 - directives, 21
 - mod_alias, 21
- redirect, 41, 66
 - RewriteRule flags, 66
- Redirect vs RewriteRule
 - recipes, 163
- REDIRECT_ prefix
 - environment variables, 59
 - internal redirect, 59
- RedirectMatch, 21, 111
 - directives, 111
 - mod_alias, 21
- referer mapping
 - RewriteMap, 106
- referer-based redirect
 - access control, 106
- Regex, 3
- regex, 173
- Regex examples, 11
- Regex Pal, 15
- regex101, 14
- regexlearn.com, 16
- RegExr, 15
- Regular expression tools, 14
- Regular expression vocabulary, 4
- Regular expressions, 3
- regular expressions, 1, 39
 - () (*grouping/capturing*), 9
 - * (*zero or more*), 7
 - + (*one or more*), 7
 - . (*dot*), 5
 - ? (*optional/non-greedy*), 8
 - \$ (*dollar sign*), 6
 - {n,m} (*repetition*), 7
 - ^ (*caret*), 6
 - [] (*character class*), 10
 - tools, 14
- REMOTE_ADDR
 - server variables, 72
- REMOTE_HOST
 - server variables, 72
- remove file extension
 - recipes, 136
- remove query string
 - recipes, 150
- Repetition, 7
- request processing
 - fixup phase, 29
 - URL-to-filename, 29
- request processing phases
 - mod_rewrite, 29
- REQUEST_FILENAME
 - server variables, 72
- REQUEST_METHOD
 - server variables, 72
- REQUEST_SCHEME
 - server variables, 72
- REQUEST_URI
 - server variables, 72
- Require not ip, 105
- response code conditional
 - logging, 119
- restrictions
 - .htaccess, 34
- reStructuredText
 - book formats, 2

- reverse proxy, 24, **173**
- reverse proxy rewrite
 - recipes, 145
- rewrite log
 - recipes, 157
- rewrite log example
 - .htaccess, 52
- rewrite logging, 46
 - .htaccess, 51
 - examples, 49
 - log entry format, 50
 - loop detection, 53
 - map lookups, 53
 - performance, 54
 - process id, 50
 - trace levels, 47
- rewrite loop
 - recipes, 152
- rewrite rule ordering
 - recipes, 155
- rewrite/access (*deprecated*), 103
- rewrite/avoid
 - access control, 103
- RewriteBase, 37, **173**
 - directives, 37
- RewriteCond, 69, **173**
 - %1, 41
 - F (*subrequest file*), 74
 - U (*subrequest URL*), 74
 - d (*directory test*), 80
 - d (*is directory*), 74
 - f (*file test*), 80
 - f (*is regular file*), 74
 - l (*is symlink*), 74
 - s (*has size*), 74
 - x (*executable*), 74
 - combining with OR, 80
 - CondPattern, 74
 - directives, 69
 - environment variables, 108
 - examples, 79
 - file attribute tests, 74
 - file existence check, 80
 - flags, 79
 - hostname matching, 79
 - HTTP_REFERER, 103
 - HTTP_USER_AGENT, 104
 - HTTPS, 107
 - HTTPS detection, 80
 - proxy decisions, 94
 - query string matching, 79
 - RewriteMap expansions, 71
 - syntax, 69
 - TestString, 71
 - TIME_HOUR, 106
 - time-based rules, 80
- RewriteCond flags
 - NC (*nocase*), 79
 - NV (*novary*), 79
 - OR (*ornext*), 79
- RewriteEngine, 31, **173**
 - directives, 31
- RewriteLog
 - recipes, 157
- RewriteMap, 34, 81, **173**
 - creating, 83
 - debugging, 53
 - deny list, 105
 - directives, 81
 - dynamic backend selection, 95
 - int:tolower, 97
 - logging, 53
 - MapName, 83
 - MapSource, 83
 - MapType, 83
 - proxy, 95
 - recipes, 167
 - referer mapping, 106
 - syntax, 83
 - types, 84
 - usage, 83
 - vhost mapping, 98
- RewriteMap CIDR
 - recipes, 169
- RewriteMap expansions
 - RewriteCond, 71
- RewriteMap internal functions
 - escape, 85
 - tolower, 84
 - toupper, 84
 - unescape, 85
- RewriteMap types
 - dbd (*SQL query*), 88
 - dbm (*hash file*), 87
 - fastdbd, 88
 - int (*internal function*), 84

- prg (*external program*), 87
- rnd (*random*), 86
- txt (*text file*), 85
- RewriteOptions, 35, **173**
 - AllowAnyURI, 35
 - AllowNoSlash, 35
 - directives, 35
 - IgnoreContextInfo, 35
 - IgnoreInherit, 35
 - Inherit, 35
 - InheritBefore, 35
 - InheritDown, 35
 - InheritDownBefore, 35
 - LegacyPrefixDocRoot, 35
 - LongURLOptimization, 35
 - MergeBase, 35
- RewriteRule, 38, **173**
 - ! (*NOT*), 40
 - .htaccess, 43
 - \$1, 41
 - %{VARNAME}, 42
 - ~user, 44
 - backreferences, 41
 - directives, 38
 - expansion order, 42
 - FLAGS, 39
 - flags, 54
 - flags summary, 44
 - home directory, 44
 - negation, 40
 - PATTERN, 39
 - pattern matching, 39
 - per-directory context, 43
 - prefix stripping, 43
 - processing order, 44
 - query string, 39
 - query string manipulation, 42
 - rule chaining, 44
 - security, 45
 - server variables, 42
 - substitution, 40
 - syntax, 38
 - TARGET, 39
 - target, 40
- RewriteRule flags, 54
 - B flag, 55
 - backrefnoplus, 56
 - BCTLS flag, 56
 - BNE flag, 56
 - BNP flag, 56
 - C flag, 57
 - chain, 57
 - CO flag, 57
 - cookie, 57
 - discardpath, 58
 - DPI flag, 58
 - E flag, 58
 - E flag; REDIRECT_ prefix, 59
 - END flag, 60
 - env, 58
 - escape backreferences, 55
 - F flag, 60
 - forbidden, 60
 - G flag, 60
 - gone, 60
 - H flag, 61
 - handler, 61
 - L flag, 61
 - last, 61
 - N flag, 63
 - NC flag, 63
 - NE flag, 63
 - next, 63
 - nocase, 63
 - noescape, 63
 - nosubreq, 63
 - NS flag, 63
 - P flag, 64
 - passthrough, 64
 - proxy, 64
 - PT (*passthrough*), 40
 - PT flag, 32, 64
 - QSA, 42
 - QSA flag, 65
 - qsappend, 65
 - QSD, 42
 - QSD flag, 65
 - qsdiscard, 65
 - QSL flag, 65
 - qslast, 65
 - R (*redirect*), 41
 - R flag, 66
 - redirect, 66
 - S flag, 66
 - skip, 66
 - T flag, 67

- type, 67
- UNC flag, 69
- UnsafeAllow3F flag, 68
- UnsafePrefixStat flag, 68

RewriteRule target

- (*dash*), 41
- absolute URL, 41
- file-system path, 40
- URL-path, 40

rnd (*random*)

- RewriteMap types, 86

robots.txt, 104

rule chaining

- RewriteRule, 44

rule order

- recipes, 155

S

S flag, 66

- RewriteRule flags, 66

ScriptAlias, 20

- directives, 20
- virtual hosts, 99

ScriptAliasMatch, 20

- mod_alias, 20

secure flag

- cookies, 58

security

- input validation, 45
- RewriteRule, 45

security considerations

- mod_rewrite, 64, 68

server context, 32, **173**

- configuration, 32

server variable, **173**

server variables, 71

- ENV, 74
- HTTP (*header lookup*), 74
- HTTP_COOKIE, 72
- HTTP_HOST, 72
- HTTP_REFERER, 72
- HTTP_USER_AGENT, 72
- HTTPS, 72
- LA-F (*filename look-ahead*), 74
- LA-U (*URL look-ahead*), 74
- QUERY_STRING, 72
- REMOTE_ADDR, 72
- REMOTE_HOST, 72

- REQUEST_FILENAME, 72
- REQUEST_METHOD, 72
- REQUEST_SCHEME, 72
- REQUEST_URI, 72
- RewriteRule, 42
- SSL, 74
- THE_REQUEST, 72

SetEnvIf query string

- recipes, 151

SetHandler

- directives, 21

SetOutputFilter, 126

skip

- RewriteRule flags, 66

Slash, 5

special characters in URLs

- recipes, 160

Sphinx

- book formats, 2

SSL

- server variables, 74

SSL/TLS

- proxy, 95

SSLProxyEngine, 95

SSLProxyVerify, 95

SSRF, 45

strip query string

- recipes, 150

subrequests, 63

Substitute directive, 121

SubstituteMaxLineLength, 121, 122

substitution, **173**

- RewriteRule, 40

syntax

- RewriteCond, 69
- RewriteMap, 83
- RewriteRule, 38

T

T flag, 67

- RewriteRule flags, 67

TARGET

- RewriteRule, 39

target

- RewriteRule, 40

TestString, **174**

- RewriteCond, 71

THE_REQUEST

- server variables, 72
 - thousands of redirects
 - recipes, 162
 - TIME_HOUR
 - RewriteCond, 106
 - time-based
 - access control, 106
 - time-based rules
 - RewriteCond, 80
 - TLS termination proxy
 - recipes, 146
 - tolower, 84
 - RewriteMap internal functions, 84
 - too many redirects
 - recipes, 152
 - tools
 - regular expressions, 14
 - toupper, 84
 - RewriteMap internal functions, 84
 - trace level
 - mod_rewrite, 48
 - trace levels
 - LogLevel, 47
 - mod_rewrite, 47
 - rewrite logging, 47
 - trailing slash redirect
 - recipes, 132
 - troubleshooting
 - proxy, 96
 - txt (*text file*)
 - RewriteMap types, 85
 - type
 - RewriteRule flags, 67
 - types
 - RewriteMap, 84
- ## U
- UNC flag, 69
 - RewriteRule flags, 69
 - UNC paths, 69
 - unescape, 85
 - RewriteMap internal functions, 85
 - UnsafeAllow3F flag, 68
 - RewriteRule flags, 68
 - UnsafePrefixStat flag, 68
 - RewriteRule flags, 68
 - URL correction
 - mod_speling, 26
 - URL encoding
 - recipes, 160
 - URL Mapping, 17
 - URL mapping, 1, 174
 - introduction, 17
 - URL-path, 174
 - URL-path, 40
 - RewriteRule target, 40
 - URL-to-filename
 - request processing, 29
 - usage
 - RewriteMap, 83
 - user agent blocking
 - access control, 104
 - user directories
 - mod_userdir, 26
 - virtual hosts, 100
 - UserDir, 26
- ## V
- version 2.2
 - Apache HTTP Server, 2
 - version 2.4
 - Apache HTTP Server, 2
 - version 2.4 configuration
 - Apache HTTP Server, 111
 - vhost mapping
 - RewriteMap, 98
 - virtual host, 174
 - virtual hosts, 23
 - Alias interaction, 99
 - CGI, 99
 - configuration, 23
 - dynamic, 97
 - IP-based, 23
 - logging, 100
 - mass hosting, 23, 97
 - mod_rewrite recipe, 97
 - mod_vhost_alias comparison, 99
 - name-based, 23
 - ScriptAlias, 99
 - user directories, 100
 - VirtualDocumentRoot, 23
 - mod_vhost_alias, 23
 - VirtualHost context, 39
 - VirtualScriptAlias, 23

W

WebSocket proxy

[recipes, 147](#)

when not to use mod_rewrite

[access control, 108](#)

Wildcard, [5](#)

wildcard subdomain redirect

[recipes, 135](#)

wss proxy

[recipes, 147](#)

www canonicalization

[recipes, 130](#)

X

X-Forwarded-Proto

[recipes, 146](#)