Sample Book Cover

Table of Contents

Table of Contents

mod_rewrite And Friends

`mod_rewrite` is one of the most powerful, and least understood, of the modules that are provided with the Apache HTTP Server. It is frequently misused to do things that can be done so much better other ways.

Thousands of examples are posted daily on various websites, showing beginners how to do things with `mod_rewrite`, and, unfortunately, the vast majority of them are wrong in various ways, subtle or grevious, due to misunderstandings of how `mod_rewrite` works, or how regular expressions work.

This book is intended to help you understand `mod_rewrite` deeply, so that you know when and how to use it, as well as when not to use it, and what to use instead.

About This Book

The first incarnation of this book, [The Definitive Guide to Apache mod_rewrite](http://drbacchus.com/book/rewrite/), was published in 2006. Since then, so much has changed that while that book is still useful, it's far from complete.

In February of 2012, Apache httpd version 2.4 was released, with a huge number of enhancements and changes. Many of the things that people have been using `mod_rewrite` for now have better solutions. Meanwhile, `mod_rewrite` itself improved quite a bit, too, and can do many new things.

This book still focuses primarily on `mod_rewrite`, but will touch on many of the surrounding topics and modules.

That said, the scope of this book has expanded (since the earlier incarnation) to include not merely URL rewriting, but also methods for munging (modifying) content, and dynamic conditional configuration. In many cases, these techniques make mod_rewrite unnecessary, or, at least, provide easier alternatives, so they fit the scope of the book very well.

These techniques include mod_substitute, mod_proxy_html, the `Define` directive, the `<If>` container, `mod_macro`, and many more. Along the way, we'll also discuss the various parts of URL mapping, the understanding of which allows you to avoid using these more complicated techniques.

How this book is organized

This book consists of 14 chapters. Depending on your level of existing expertise, some of them can be safely skipped.

Chapter 1 - Regular Expressions - This chapter gives an introduction to regular expressions, which are the language of `mod_rewrite`.

Chapter 2 - URL Mapping - URL rewriting is a portion of a larger topic called URL mapping - the process by which Apache httpd translates a requested URL into an actual resource that it will serve.

Chapter 3 - An introduction to `mod_rewrite` - covering some of the configuration directives that need to be set up before you start rewriting.

Chapter 4 - RewriteRule - The `RewriteRule` directive is the one you'll be using most often. This chapter covers its syntax and usage.

Chapter 5 - Rewrite Logging - The rewrite log is a great debugging tool, and also a good way to learn about how `mod_rewrite` thinks about things.

Chapter 6 - RewriteRule flags - Flags modify the behavior of `RewriteRule` . They've been introduced in the previous chapter, but this chapter covers each flag in detail, with examples.

Chapter 7 - RewriteCond - `RewriteCond` allows you to put conditions on the running of a particular `RewriteRule` .

Chapter 8 - RewriteMap - The `RewriteMap` directive allows you to craft your own `RewriteRule` logic and lookup tables.

Chapter 9 - Proxying with mod_rewrite - `RewriteRule's `[P]` flag lets you pass a request through a proxy. This chapter digs into that in greater detail.

Chapter 10 - Virtual hosts with mod_rewrite - Using `RewriteRule` to manage virtual hosts.

Chapter 11 - Access control with mod_rewrite - Using `RewriteRule` to control or restrict access to resources.

Chapter 12 - Configurable Configuration - New in version 2.4 of the web server is a class of directives that let you add intelligence and request-time decisions to the configuration. These techniques replace many of the things that people used to use `mod_rewrite` for.

Chapter 13 - Content Modification Modules - In this chapter, we discuss rewriting content sent to the client, which is not something that `mod_rewrite` does.

Chapter 14 - Recipes - Recipes, and detailed discussions of them, addressing common problems and solutions.

Other Sources of Wisdom

A brief word about the documentation. The official docs, at http://httpd.apache.org/docs/current, are great, and are the work of many dedicated people. I'm one of many. This book is intended to augment those docs, and not replace them. If it appears sometimes that I have copied shamelessly from the documentation, I humbly ask you to remember that I participated in writing those docs, and the edits flowed both directions — that is, sometimes it was the docs that shamelessly copied from the book.

This book does **not** attempt to be a comprehensive book about the Apache web server. For that, I encourage you to look the documentation and also at my other book, **Apache Cookbook, Third Edition**, by Rich Bowen and Ken Coar, which should be available around the same time that this book is published.

You should also acquire a copy of Jeffrey Friedl's excellent book, **Mastering Regular Expressions** - http://shop.oreilly.com/product/9780596528126.do While the book is several years old, it is still the best book on the topic.

Technical details

This book was written in Markdown, using vim — <https://www.vim.org/> — and built using gitbook — https://toolchain.gitbook.com/.

Previous incarnations were written in LaTeX, ReStructuredText, AsciiDoc, and who knows what else. There always seems to be a new book format out there. It's exhausting.

You can always obtain the most recent version of the book at http://mod-rewrite.org/, and you'll usually be able to buy a fairly recent version in the Amazon Kindle store. Some day, there will hopefully be a printed version, too.

Contact information, and errata reporting

If you'd like to get involved in the creation of this book, or if you'd like to tell me about something that needs fixed, Go to GitHub - https://github.com/rbowen/mod_rewrite_book - and either submit pull requests or open a ticket. If you don't know what that means, you are welcome to submit errata to <rbowen@rcbowen.com>, and some day there will be a handy way to do this on the website. Not today.

This book is a work in progress. If you purchased the book in electronic form, you should be eligible to receive updates from wherever you bought it. If you're not, send me your email receipt <rbowen@rcbowen.com>, and I'll send you an updated version.

About the Author

Rich Bowen has been involved on the Apache http server documentation since about 1998. He is also the author of **Apache Cookbook**, and **The Definitive Guide to Apache mod_rewrite**. You can frequently find him in \#httpd, on `irc.freenode.net` . under the name of `DrBacchus` or `rbowen` .

Rich works at Red Hat, in the OSAS (Open Source and Standards) group, where he is an Open Source Community Manager. See http://community.redhat.com/ for details.

He lives in Lexington, Kentucky, with his wife and kids.

Acknowledgements

Thanks to `fajita` , and the other regulars on #httpd (on the `irc.freenode.net` network). `fajita` is my research assistant, and knows more than everyone else on the channel put together. And the folks on #ahd who keep me sane. Or insane. Depending on how you measure. A warm hog to each of you.

None of this would be possible without `mod_rewrite` itself, so a big thank you to Ralf Engelschall for creating it, and all the many people who have worked on the code and documentation since then.

Finally, a thank you to my muses, Rhi, Z, and E.

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

Table of Contents

Chapter 1: Regular Expressions

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

— Rudyard Kipling
The Elephant's Child

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** http://regex.info/book.html by Jeffrey Friedl.

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.

The Building Blocks

Regular expressions are a means to describe a text pattern (technically, it's any data, but in the context of Apache httpd, we're primarily interested in text as it appears in URLs), 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.

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.

Table 1. Regular Expression Vocabulary

Character	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 Escaping characters 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 Greedy Matching.
`{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

	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.

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 parenthesis 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.

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.

Table 2. Metacharacters

Character	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 (`\babc`) and/or end (`abc\b`) of a word, thus `\bcow\b` will match cow but not cows, but `\bcow` will match cows.
\B	Not a word boundary. Match any character(s) NOT at the beginning(`\Babc`) and/or end (`cow\B`) of a word, thus `\Bcow\B` will match scows but not cows, but `cow\B` will match coward.
\t	Match a tab character
\n	Match a newline character
\x	Matches a character with a particular hex code. For example, `\x5A` 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.

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`_

Table 3. Anchor examples

Example	Meaning
`^/`	This matches any string that starts with a slash
`.jpg$`	This pattern matches any string that ends with .jpg.
`/$`	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.
`^$`	Matched 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 allow 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.

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 "giraffffeeeeee".

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

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 "giraeeeee", 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 `#`.

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.

Table 4. Repition 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 repitition 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.

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.

Table 5. Examples of greedy matching

Pattern	Matches
`/(.*)/`	one/two/three
`/(.*?)/`	one

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.

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-greed behavior will follow once we have learned about backreferences.

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".

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 be more obvious later, once we start looking at the RewriteRule directive in :ref:`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.

TODO Recommend the correct regex tool

It is instructive to acquire a tool such as Regex Coach, or Rebug, mentioned 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.

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.

Negation

a. index:: Negation

b. index:: !

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 :ref:`Proxying with mod_rewrite`.

Regex examples

a. index:: Examples

b. index:: 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.

Email address

a. index:: 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:]._-]@(\.)[: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.

Phone number

a. index:: 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 though 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 parenthesis, 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, say, an underscore, you could 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.

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.

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 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.

Matching a directory

a. index:: 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`.

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.

Regex tools

TODO Ensure that these tools all still exist.

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. You'll find that most of the really good tools for regular expression development come out of the Perl community, where regular expressions are particularly popular, and tend to get used in almost every program.

Regex Coach

Regex Coach is available for Windows and Linux, and can be downloaded from http://www.weitz.de/regex-coach. Regex Coach allows you to step through a regular expression and watch what it does and does not match. This can be extremely instructive in learning to write your own regular expressions.

TODO

SCREENSHOT

Regex Coach is free, but it is not Open Source.

Reggy

Reggy is a Mac OS X application that provides a simple interface for crafting and testing regular expressions. It will identify what parts of a string are matched by your regular expression.

Reggy is available at http://code.google.com/p/reggy/ and is licensed under the GPL.

TODO

SCREENSHOT

pcretest

pcretest is a command-line regular expression tester that is available on most distributions of Linux, where it is usually installed by default.

In addition to simply telling you whether a particular string matched or not, it will also tell you what each of the various backreferences will be set to.

In the SCREENSHOT you can see what each of the various backreferences will be set to once the regular expression has been evaluated.

TODO: Screen shot

Visual Regexp

Visual Regexp, available at http://laurent.riesterer.free.fr/regexp/, has more features than the options listed above, and might be a good option once you have mastered the basics of regular expressions and are ready to move onto something a little more sophisticated. It shows backreferences, and offers a wide variety of suggestions to help build a regex.

Visual Regexp is available as a Windows executable or as a Tcl/Tk script.

TODO

SCREENSHOT

Regular Expression Tester

Rather than being a stand-alone application like the others listed above, this is a Firefox plugin. It's available at https://addons.mozilla.org/en-US/firefox/addon/2077, and, of course, requires Firefox to work.

Online tools

a. index:: Online regex testers

In addition to these tools, there are many online tools, which you can use without having to download or install anything. These are of a wide variety of feature sets and quality, so I'd encourage you to shop around a little to find one that seems to work well. These appear and disappear on a weekly basis, and so I can't promise that these sites will still be available at the time that you read this, but here are some that are worth mentioning at the time of writing:

RegExr

a. index:: RegExr

http://gskinner.com/RegExr/ - Includes a variety of pre-defined character classes, and the ability to save your regular expressions for later reference. Requires Javascript to use.

Regex Pal

a. index:: Regex Pal

http://regexpal.com/ - Less full-featured than RegExr, but sufficient for the purpose of crafting and testing regular expressions for the purpose of mod_rewrite, which doesn't require replace functionality or multi-line matches.

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.

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 http://www.perldoc.com/perl5.8.0/pod/perlre.html or by typing `perldoc perlre` at your command line, and the PCRE documentation, which you can find online at http://pcre.org/pcre.txt. 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.

Table of Contents

Chapter 2: URL Mapping

In this chapter, we'll discuss the various ways that the Apache http server 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.

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. Indeed, mod_rewrite spans several chapters of this book, and has an entire Part all its own, part mod_rewrite.

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.

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.

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

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.

Automatic directory listings

The module mod_autoindex serves a file listing for any directory that doesn't contain a DirectoryIndex file. (See DirectoryIndex <directoryindex>.)

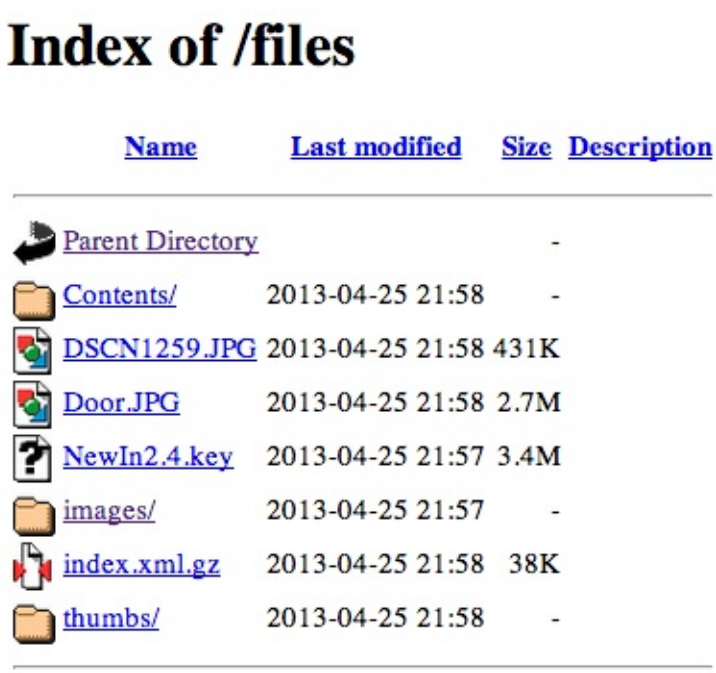
To permit directory listings, you must enable the Indexes setting of the Options directive:

```
Options +Indexes
```

See the documentation of the Options http://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.



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

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

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 http://httpd.apache.org/docs/current/howto/cgi.html

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
```

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 by 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>
```

Virtual Hosts

Rather than running a separate physical server, or separate instance of httpd, for each website, it is common practice 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.

Proxying

TODO

mod_actions

TODO

mod_imagemap

TODO

mod_negotiation

TODO

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 …

TODO

Table of Contents

Chapter 5: Rewrite Logging

Exactly how you turn on logging for mod_rewrite will depend on what version of the Apache http server you are running. Logging got some updates in the 2.4 release of the server, and the rewrite log was one of the changes that happened at that time.

If you're not sure what version you're running, you can get the `httpd` binary to tell you with the `-v` flag:

```
httpd -v
```

As with any other logging, the log file is opened when the server is started up, before the server relinquishes its root privileges. For this reason, the `RewriteLog` directive may not be used in `.htaccess` files, but may only be invoked in the server configuration file.

2.2 and earlier

Prior to httpd 2.4, the way to enable mod_rewrite logging is with the `RewriteLog` and `RewriteLogLevel` directives.

The `RewriteLog` directive should be set to the location of your rewrite log file, and the `RewriteLogLevel` is set to a value from 0 to 5 to indicate the desired verbosity of the log file, with 0 being no log entries, and 5 being to log every time mod_rewrite even thinks about doing something.

You'll often find advice online suggesting that `RewriteLogLevel` be set to 9 for maximum verbosity. Numbers higher than 5 don't make it more verbose, but they also don't harm anything.

```
RewriteLog logs/rewrite.log
RewriteLogLevel 5
```

2.4 and later

In the 2.4 version of the server, many changes were made to the way that logging works. One of these changes was the addition of per-module log configurations. This rendered the `RewriteLog` directive superfluous. So, from 2.4 on, rewrite logging is enabled using the `LogLevel` directive, specifying a `trace` log level for mod_rewrite.

```
LogLevel info rewrite:trace6
```

Rewrite log entries will now show up in the main error log file, as specified by the `ErrorLog` directive.

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 info rewrite:trace6

# Or, in 2.2
```

```
# RewriteLog Level 5
# RewriteLog /var/log/httpd/rewrite.log
```

This ruleset says "If it's not already `index.php`, rewrite it to `index.php`.

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

```
[Thu Sep 12 20:22:13.363463 2013] [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 12 20:22:13.363510 2013] [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 12 20:22:13.363525 2013] [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 12 20:22:13.363533 2013] [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 12 20:22:13.363542 2013] [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 12 20:22:13.575877 2013] [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

[Thu Sep 12 20:22:13.575920 2013] [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 12 20:22:13.575935 2013] [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 12 20:22:13.575943 2013] [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 12 20:22:13.575955 2013] [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
```

This is an entry from a 2.4 server, and contains a few elements that will be missing from rewrite log entries for 2.2 and earlier.[1]

Note that I've inserted linebreaks between each log entry for legibility. And speaking of legibility, let's consider one single log entry to see what the various components mean before we go any further.

Let's look at the first log entry.

:

```
[Thu Sep 12 20:22:13.363463 2013] [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 we'll break it down one field at a time.

`[Thu Sep 12 20:22:13.363463 2013]`
The date and time when the event occurred.

`[rewrite:trace2]`
The name of the module logging, and the loglevel at which it is logging. This is 2.4-specific

`[pid 1879]`
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 we 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.[2]

RewriteRules in .htaccess files - An example

We'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 our 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 info rewrite:trace6

# Or, in 2.2
# RewriteLog Level 5
# RewriteLog /var/log/httpd/rewrite.log
```

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, of course, is that this is much longer than what we 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 special 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 perdir 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 we did the first time through, now we're looking at the string `example`. While this may seem trivial at this point, 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 perdir 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.

---

1. Future editions of this book will contain full examples from a 2.2 server, for those still running that version.
2. Future editions of this book will contain an appendix in which several log traces are explained in exhaustive detail. I can hardly wait.

Table of Contents

Chapter 6: RewriteRule Flags

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]
```

TODO Rewrite Flags should be a separate chapter

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

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.

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.

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.

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:

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.

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.

Secure

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

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.

Example

Consider this example:

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

In the example give, 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 '/').

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.

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 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]
```

END

Although the flags are presented here in alphabetical order, it makes more sense to go read the section about the L flag first (ref{lflag}) 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 (htaccess) context.

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

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.

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.

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` referrers to the captured match within parenthesis of the regular expression.

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, it is important to have some understanding of how the rules are processed. The simplified form of this is that once the rules have been processed, the rewritten request is handed back to the URL parsing engine to do what it may with it. It is possible that as the rewritten request is handled, the .htaccess file or `<Directory>` section may be encountered again, and thus the ruleset may be run again from the start. Most commonly this will happen if one of the rules causes a redirect - either internal or external - causing the request process to start over.

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.

An alternative flag, `[END]`, can be used to terminate not only the current round of rewrite processing but prevent any subsequent rewrite processing from occurring in per-directory (htaccess) context. This does not apply to new requests resulting from external redirects.

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.

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 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).

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]
```

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.

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.

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 ref{chapter_proxy} for a more thorough treatment of proxying.

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 -.

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.

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.

This flag is available in version 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.

R - redirect

Use of the `[R]` flag causes a 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.

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]

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 ref{if}.)

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]
```

Please note that this is a trivial example, and 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.

Table of Contents

Chapter 7: RewriteCond

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 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.

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 ⇐ N ⇐ 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 ⇐ N ⇐ 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 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 a HTTP header is used in a condition this header is added to the Vary header of the response in case the condition evaluates to 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.

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.

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

'=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.

'⇐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).

'>=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).

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.

'-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.

'-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.

'-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.

'-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.

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.

'-f' (is regular file)
Treats the TestString as a pathname and tests whether or not it exists, and is a regular file.

'-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!

'-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.

'-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.

'-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!

'-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.

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 a 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.

Examples

Query Strings .. index
rewritemap_int """""""""

Table of Contents

Chapter 8: RewriteMap

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`.

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.

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}}
```

RewriteMap Types

There are a number of different map types which may be used in a `RewriteMap`.

int

An `int` map type is an internal function, pre-defined by `mod_rewrite` itself. There are four such functions:

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]
```

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]
```

escape

unescape

txt

A `txt` map defines a one-to-one mapping from argument to target.

rnd

A `rnd` map will randomly select one value from the specified text file.

dbm

prg

dbd

Table of Contents

Chapter 9: Proxies and mod_rewrite

Table of Contents

Chapter 10: Virtual hosts and mod_rewrite

Table of Contents

Chapter 11: Access control with mod_rewrite

Table of Contents

Chapter 12: Conditional Configuration

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.

Match Directives

FilesMatch, RedirectMatch, etc.

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.)

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>
```

<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:

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>
```

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
    </If>
```

```
</FilesMatch>
```

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.

mod_proxy_express

mod_vhost_alias

Conditional logging

env=

Per-module logging

Per-directory logging

Piped logging

Table of Contents

Chapter 13: Content Munging

While mod_rewrite modifies aspects of the HTTP request - most commonly the REQUEST_URI, sometimes you want to modify the content which is served to the client. There are several modules that do this, which can be used in a variety of circumstances.

We're going to look at three of these modules, and then at Filters in general.

mod_substitute

mod_sed

mod_proxy_html

Filters

Table of Contents

Chapter 14: Recipes

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.

TODO

This book is a work in progress, and I expect it to remain such for years to come. This is the place to check to see if you've purchased the latest version, and what changed from one version to another.

While the version number starts with 0.something, you can expect that there's quite a bit of work yet to do. Once it is 1.something, you can expect that changes will be fairly minor. I think. We'll see.

TODO

- Standardize how we display example
- Write mod_rewrite chapter(s)
- Write mod_rewrite logging and debugging chapter
- Write mod_rewrite examples chapter
- Write Content Munging chapter
- Write Conditional Configuration chapter
- Provide issue tracker where people can log errata
- Convert all LaTeX to RST
- Verify all desired formats (pdf, html, epub)
- Automated publishing tools
- Asciidoctor-pdf doesn't do footnotes right. Either they need to fix this, or I need to remove all footnotes.
- Update/Replace the second on regex testing tools, since these appear and vanish pretty quickly.

REVISION HISTORY

- 0.00 Started March 7, 2013. Started TOC and a little of the initial text. Published HTML version to website at http://rewrite.rcbowen.com/
- 0.01 March 12, 2013. Initial publish to Amazon.com in Kindle form.
- 0.02 March 18, 2013. Munged the TOC around a bit to make the chapters less crowded. Will end up with some sparse chapters initially. So what.
- 0.03 March 18, 2013. Added a bunch about flags. Completed reorg of TOC. I hope.
- ...
- 0.12 April 16, 2013. Started RewriteMap stuff, and various other tweaks and fixes.
- 0.15 August 10, 2013. Attended Flock. Decided to convert all LaTeX to rST instead. Many benefits, but quite a bit of work. Should have a rebuild in the next few days.
- 0.20 August 12, 2013. Completed conversion from LaTeX to rST. I'm sure there's still some orts here and there, but it's good enough to tag.
- 0.30 - Christmas 2017. Yet another conversion, this time, to ASCIIdoc. Borrowed tools and templates from https://github.com/akosma/eBook-Template to get started. Website has been moved to http://mod-rewrite.org. We'll start publising it there again once we have a shippable version.
- 0.31 - Christmas 2018. Will the format changing never end? Converted to MarkDown and GitBook. https://toolchain.gitbook.com/ But I'm starting to remember that I rejected GitBook because it doesn't seem like there's a way to generate an index easily.
- 0.32 - The brief experiement with going back to Markdown abandoned, since Gitbook supports asciidoc. I'm going to focus on writing, and figure out indexing at some later date.