Table of Contents

`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 13 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 - mod_rewrite - 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.

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 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 beatiful. And so that's all right, Best Beloved, do you see?

[[Chapter_regular_expressions]]

Regular Expressions

indexterm:[Regular expressions] indexterm:[Introduction to regular expressions] indexterm:[Regex]

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.

indexterm:[Mastering Regular Expressions by Jeffrey Friedl] indexterm:["Friedl, Jeffrey"]

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.

indexterm:[Regular expression vocabulary] [options="header"] .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 <> 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 <>. | `{n,m}` | Indicates that the previous thing should match at least n, and not more than m times. For example, `a{2,7}` matches at least 2, and not more than 7, occurrences of the letter a | `( )` | Provides grouping and capturing functions. Grouping means treating more than one character as though they were a single unit. You can apply repetition characters to a group created in this way. Capturing means remembering the thing that matched, so that we can use it again later. This is called a 'backreference.' | `[ ]` | Called a "character class," this matches only one of the contained characters. For example, `[abc]` matches a single character which is either a or b or c. | `^` | Negates a match within a character set. (Remember that outside of a character class, it means something else. See above.) Thus, `[^abc]` matches a single character which is neither a nor b nor c. | `!` | Placed on the front of a regular expression, this means "NOT". That is, it negates the match, and so succeeds only if the string does not match the pattern. |======================

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

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

[[Wildcard_character]]

Matching anything

indexterm:[Wildcard] indexterm:[.]

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]]

Escaping characters

indexterm:[Escape] indexterm:[Metacharacters] indexterm:[Backslash] indexterm:[Slash]

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.

indexterm:[Metacharacter table] [options="header"] .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

indexterm:[Anchors] indexterm: indexterm:[$]

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 _`

indexterm:[Anchor examples] [options="header"] .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

indexterm:[+] indexterm:[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

indexterm:[*] indexterm:[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

indexterm:["{n,m}"] indexterm:[Repetition]

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.

[options="header"] .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]]

Greedy Matching

indexterm:[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.

indexterm:[Examples of greedy matching] indexterm:[Greedy matching,examples] [options="header"] .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

indexterm:[Optional matching] indexterm:["?"]

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 <>.

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

Grouping and capturing

indexterm:[Grouping] indexterm:[Capturing] indexterm:[( )]

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

indexterm:[Backreferences] indexterm:[$1] indexterm:[%1]

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 <> 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

indexterm:[[ ]] indexterm:[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

.. index:: Negation .. 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

.. index:: Examples .. 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

.. 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:]._-]@([: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

.. 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

.. 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]]

Regex tools

indexterm:[Regular expression tools] indexterm:[Tools,regular expressions]

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

indexterm:[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

indexterm:[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

indexterm:[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

indexterm:[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

indexterm:[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

.. 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

.. 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

.. 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.

[[Chapter_url_mapping]]

URL Mapping

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

[[introduction-to-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

```
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]]
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 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]] 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.

image:autoindex1.png[AutoIndex]

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]]
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]]
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:

[cols="",options="header",] |== |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]] Location ≃~~~

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

sections.

sections operate completely outside the filesystem. This has several consequences. Most importantly, 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> ....

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> ....

When to use Use to apply directives to content that lives outside the filesystem. For content that lives in the filesystem, use and . An exception is , 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>

would match URLs that contained the substring /extra/data or /special/data. The directive behaves identically to the regex version of , 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>

The 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:

SetHandler server-status Require host example.com

</Location>

[[virtual-hosts]]

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]]

Proxying

TODO

[[mod_actions]]

mod_actions

TODO

[[mod_imagemap]]

mod_imagemap

TODO

[[mod_negotiation]]

mod_negotiation

TODO

[[file-not-found]]

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