APPENDIX A

Using Regular Expressions in Apache

A number of the Apache web server's configuration directives permit (or require!) the use of what are called *regular expressions*. Regular expressions are used to determine if a string, such as a URL or a user's name, matches a pattern.

There are numerous resources that cover regular expressions in excruciating detail, so this appendix is not designed to be a tutorial for their use. Instead, it documents the specific features of regular expressions used by Apache—what's available and what isn't. Even though there are quite a number of regular expression packages, with differing feature sets, there are some commonalities among them. The Perl language, for instance, has a particularly rich set of regular expressions, which have been mostly available in the Apache regex library since version 2.0 of the server.

Regular expressions, as mentioned, are a language that allows you to determine if a particular string or variable looks like some pattern. For example, you may wish to determine if a particular string is all uppercase, or if it contains at least three numbers, or perhaps if it contains the word "monkey" or "Monkey." Regular expressions provide a vocabulary for talking about these sort of tests. Most modern programming languages contain some variety of regular expression library, and they tend to have a large number of things in common, although they may differ in small details.

Apache 1.3 uses a regular expression library called *hsregex*, so called because it was developed by Henry Spencer. Note that this is the same regular expression library that has been used in the *egrep*.

Apache 2.0 uses a somewhat more full-featured regular expression library called Perl Compatible Regular Expressions (PCRE), so called because it implements many of the features available in the regular expression engine that comes with the Perl programming language. Although this Appendix does not attempt to communicate all the differences between these two implementations, you should know that *hsregex* is a subset of PCRE, as far as functionality goes, so everything you can do with regular expressions in Apache 1.3, you can do in 2.0, but not necessarily the other way around.

To grossly simplify, regular expressions implement two kinds of characters. Some characters mean exactly what they say (for example, a `G` appearing in a regular expression will usually mean the literal character G), whereas some characters have special significance (for example, the period [.] will match any character at all—a wildcard character). Regular expressions can be composed of these characters to represent (almost) any desired pattern appearing in a string. (Recursive patterns, for example, are not possible.)

What Directives Use Regular Expressions?

Two main categories of Apache directives use regular expressions. Any directive with a name containing the word *Match*, such as FilesMatch, can be assumed to use regular expressions in its arguments. And directives supplied by the module *mod_rewrite* use regular expressions to accomplish their work.

For more about *mod_rewrite*, see Chapter 5.

Something*Match* directives each implement the same functionality as their counterpart without the *Match*. For example, the *RedirectMatch* directive does essentially the same thing as the *Redirect* directive, except that the first argument, rather than being a literal string, is a regular expression, which will be compared to the incoming request URL.

Regular Expression Basics

To get started in writing your own regular expressions, you'll need to know a few basic pieces of vocabulary, such as shown in Table A-1 and Table A-2. These constitute the bare minimum that you need to know. Although this will hardly qualify you as an expert, it will enable you to solve many of the regex scenarios you will find yourself faced with.

Table A-1. A basic regex vocabulary

Character	Meaning
.	Matches any character. This is the wildcard character.
+	Matches one or more of the previous character. For example, M+ would match one or more Ms; ".+" would match one or more characters of any kind.
*	Matches zero or more of the previous character. For example, M* would match zero or more Ms. This means that it will not only match M, MM, and MMM, but it will also match a string that doesn't have any Ms in it at all.
?	Makes the previous character optional. For example, the regular expression monkeys? will match a string containing either monkey or monkeys.
^	Indicates that the following characters must appear at the beginning of the string being tested. Thus, a regular expression of ^zim requires that the string being tested start with the characters zim. ^ is referred to as an anchor, because it anchors the match to the beginning of the string. In the context of a character class (see below), the ^ character has another special meaning.

Character	Meaning
$	Indicates that the characters to be matched must appear at the end of the string. Thus, a regular expression of `gif$` requires that the string being tested end with the characters `gif`. $ is referred to as an anchor, because it anchors the match to the end of the string.
\	Escapes the following character, meaning that it removes the "specialness" of the character. For example, a pattern containing `\.` would match a literal `.` character, since the `\.` removes the special meaning of the `.` character.
[]	Character class. Match one of the things contained in the square brackets. For example, `[abc]` will match either an `a`, or `b`, or `c`. `[abc]+`, on the other hand, would match a sequence of `a`'s, `b`'s, and `c`'s, or any combination of them. Note that within a character class, the ^ character doesn't have its normal anchor status but means any character *except* those in the class. Thus, a character class of `[^abc]` will match any character that is *not* an `a`, `b`, nor `c`. A character class containing a `-` between two characters means an entire range of characters. For example, the character class `[a-q]` means all of the lowercase letters starting from `a` and ending with `q`. `[a-zA-Z]` would be all uppercase, and all lowercase letters. In addition to character classes that you form yourself, there are a number of special predefined character classes to represent commonly used groups of characters. See Table A-2 for a list of these predefined character classes.
()	Groups a set of characters together. This allows you to consider them as a single unit. For example, you could apply a `+` or `?` to an entire group of characters, rather than just a single character. The expression `(monkeys)?`, for example, would make the entire word `monkeys` an optional part of the match. In some regular expression libraries, the `( )` characters also capture the contents of the match so that they can be used later. (The regex libraries used by the Apache HTTP server *always* do this.)

Two Common Mistakes

A common pitfall when constructing regular expression (also called "RE" or "regex" [pronounced "rej-eks"]) patterns is overlooking that * matches *zero* or more occurrences. This means that a pattern of **^first field:*lastfield$**, intended to match things like **first field:blah:foo:blah:lastfield** will *also* match **firstfieldlastfield**. Often you actually want to use a + meaning *one* or more occurrences.

Another fact often forgotten is that pattern characters that match a variable number of occurrences—*, +, and ?—apply to the single pattern expression they immediately follow. And unless you've grouped several together with parentheses, that usually means a single character. **foo+** will match **foo** and **foooooo**, *not* **foofoofoofoo**. To match the latter, group the characters like this: **(foo)+**.

Table A-2. Predefined regular expression character classes

Character class	Meaning
\d	Any decimal digit ("0" through "9")
\D	Any character that is *not* a decimal digit

Character class	Meaning
\s	Any whitespace character except VT (vertical TAB). This matches space, HT, CR, LF, and FF. Note that this differs from the POSIX class `[:space:]` described elsewhere in this table, which matches all of those characters *and* VT.
\S	Any nonwhitespace character (that is, anything that doesn't match `\s`)
\w	Any "word" character; that is, an underscore or any character with an ordinal value less than 255 that is a letter or decimal digit. This has restrictions and ideosyncrasies which are affected by the current locale. See the PCRE documentation for details.
\W	
[:alnum:]	Any alphanumeric character
[:alpha:]	Any alphabetical character
[:blank:]	A space or horizontal tab
[:cntrl:]	A control character
[:digit:]	A decimal digit
[:graph:]	A nonspace, noncontrol character
[:lower:]	A lowercase letter
[:print:]	Same as graph, but also space and tab
[:punct:]	A punctuation character
[:space:]	Any whitespace character, including newline and return
[:upper:]	An uppercase letter
[:xdigit:]	A valid hexadecimal digit

[ ] and [: :]

The POSIX classes (the ones using `[: :]`) are shorthand names for multiple matching characters, and the `[:` and `:]` are part of their syntax. As classes, they can only be used within a class expression (*i.e.*, between `[` and `]`), which can make things look a little weird and confusing. For instance, to match any single hexadecimal digit, you'd use:

```
[[:xdigit:]]
```

To match more than one, you'd use:

```
[[:xdigit:]]+
```

To match any hexidecimal digit *or* whitespace character:

```
[[:xdigit:][:space:]]
```

or maybe

```
[[:xdigit:]\s]
```

To match any character *except* a hexidecimal digit, use:

```
[^[:xdigit:]]
```

(The PCRE implementation of POSIX classes also allows

```
[[:^xdigit:]]
```

but that's an extension syntax and not universally supported.)

Examples

The previous concepts can best be illustrated by a few examples of regular expressions in action.

Redirecting several URLs

We'll start with something fairly simple. In this scenario, we're getting a new Web server to handle the customer support portion of our Web site. So, all requests that previously went to *http://www.example.com/support/* will now go to the new server, *http://support.example.com/*. Ordinarily, this could be accomplished with a simple *Redirect* statement, but it appears that our Web site developer has been careless and has been using *mod_speling* (see Recipe 5.9), so there are links throughout the site to both *http://www.example.com/support/* and to *http://www.example.com/Support/*, which would actually require not one but two *Redirect* statements.

So, instead of using the two *Redirect* statements, we will use the following one *RedirectMatch* directive:

```
RedirectMatch ^/[sS]upport/(.*) http://support.example.com/$1
```

The square brackets indicate a character class, causing this one statement to match requests with either the upper- or lowercase s.

Note also the ^ on the front of the argument, causing this directive to apply only to URLs that *start* with the specified pattern, rather than URLs that simply happen to contain that pattern somewhere in them.

The parentheses on the end of the regular expression capture the remainder of the reuqested URI, putting it in the variable $1 for later use. We use it in the redirect target to retain the remainder of the requested URI.

Catching common misspellings

While watching the logfiles, we see that a number of people are misspelling support as suport. This is easily fixed by slightly altering our *RedirectMatch* directive:

```
RedirectMatch ^/[sS]upp?ort/(.*) http://support.example.com/$1
```

The ? makes the second p optional, thus catching those requests that are misspelled and redirecting them to the appropriate place anyway.

For More Information

By far the best resources for learning about regular expressions are Jeffrey Friedl's book *Mastering Regular Expressions* and Tony Stubblebind's book *Regular Expression Pocket Reference*, both published by O'Reilly. They cover regular expressions in many languages, as well as the theory behind regular expressions in general.

For a free resource on regular expressions, you should see the Perl documentation on the topic. Just type `perldoc perlre` on any system that has Perl installed. Or you can view this documentation online at *http://perldoc.perl.org/perlre.html*. But be aware that there are subtle (and not-so-subtle) differences between the regular expression vocabulary of Perl and that of the PCRE libraru (the one used by Apache). These differences are described on the Web at *http://www.pcre.org/pcre.txt*.