mod_rewrite. Podręcznik administratora

Autor: Rich Bowen
Tłumaczenie: Marek Pałczyński, Michał Rogalski
ISBN: 978-83-246-0465-4
Tytuł oryginału: The Definitive Guide to Apache mod_rewrite
Format: B5, stron: 160

Moduł mod_rewrite, nazywany często „scyzorykiem armii szwajcarskiej", to potężne narzędzie administratorów serwerów WWW, które uruchomiono w oparciu o Apache. Za jego pomocą można w dowolny niemal sposób manipulować adresami URL wpisywanymi przez użytkowników w przeglądarce internetowej. Mod_rewrite, oparty na dyrektywach i wyrażeniach regularnych, pozwala na zmianę postaci adresów URL w oparciu o zmienne środowiskowe, nagłówki HTTP i wiele innych warunków. Ogromne możliwości mod_rewrite sprawiają, że jest on określany jako „równie skomplikowany i tajemniczy jak magia woodoo".

Celem książki „mod_rewrite. Podręcznik administratora" jest przybliżenie możliwości tego modułu administratorom serwerów. Przedstawia ona zarówno zagadnienia podstawowe, czyli obszar zastosowań modułu oraz zasady wykorzystania wyrażeń regularnych, jak i zagadnienia zaawansowane, takie jak tworzenie serwerów wirtualnych, kontrola dostępu i użycie dyrektyw warunkowych. Czytając tę książkę, dowiesz się, kiedy należy stosować mod_rewrite, jak go instalować i konfigurować, jak pracować z mechanizmem przepisywania oraz w jaki sposób wyszukiwać i usuwać błędy w dyrektywach modułu.

- Podstawowe wiadomości o mod_rewrite
- Wyrażenia regularne
- Procedura instalowania modułu mod_rewrite
- Korzystanie z dyrektywy RewriteRule
- Przepisywanie warunkowe
- Zewnętrzny mechanizm odwzorowania adresów
- Dynamiczne tworzenie serwerów wirtualnych
- Współpraca modułu mod_rewrite z modułem mod_proxy

Przekonaj się, że mod_rewrite jest mniej skomplikowany od formuły uzyskania złota z ołowiu

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Wyrażenia regularne

Działanie modułu mod_rewrite bazuje na definicjach wyrażeń regularnych zgodnych z wyrażeniami języka Perl (PCRE — *Perl Compatible Regular Expression*). Z tego względu osoby, które dzięki tej książce chcą poszerzyć swoją wiedzę na temat modułu mod_rewrite, muszą najpierw zapoznać się z podstawami wyrażeń regularnych. Znajomość zaawansowanych mechanizmów działania wyrażeń regularnych nie jest tutaj konieczna, ale niezbędne jest przynajmniej poznanie ich składni. Nie zaszkodzi również przygotowanie sobie podręcznego słownika wyrażeń regularnych.

Wszystkie potrzebne informacje zostaną przedstawione w niniejszym rozdziale. Trzeba jednak pamiętać, że istnieje wiele szczegółowych opracowań związanych z tym tematem. Wyrażenia regularne są bowiem obszernym zagadnieniem, często opisywanym w różnych dokumentach. Na szczególną uwagę zasługuje książka Jeffreya Friedla pt. *Wyrażenia regularne* (Helion, 2001). Jest to rzetelna, świetnie napisana i wyczerpująca publikacja.

Celem niniejszego rozdziału jest przedstawienie podstawowych elementów składowych wyrażeń regularnych — podstawowych symboli — oraz omówienie niektórych zaawansowanych technik przygotowywania własnych wyrażeń regularnych oraz analizowania wyrażeń definiowanych przez kogoś innego.

Osoby, które znają składnię wyrażeń regularnych, mogą pominąć ten rozdział.

Podstawowe elementy składowe

Wyrażenia regularne są sposobem na opisanie wzorców ciągów tekstowych (w zasadzie mogą opisywać dane dowolnego rodzaju, ale zazwyczaj odnoszą się do tekstu), umożliwiających wyszukanie określonego ciągu w bloku danych. Najlepszym sposobem analizowania jakiegokolwiek wyrażenia regularnego jest interpretowanie pojedynczych znaków. Trzeba więc wiedzieć, co każdy ze znaków reprezentuje.

Znaki te są podstawowymi elementami składowymi wyrażeń regularnych. Osoby, które nie znają składni wyrażeń regularnych, powinny włożyć w to miejsce książki zakładkę, gdyż będą musiały niejednokrotnie odwoływać się do prezentowanych tu informacji (dopóki nie nabędą biegłości w korzystaniu z opisywanych symboli). Tabela 2.1 jest kluczem do przekształcenia pozornie przypadkowego zbioru znaków we wzorzec dopasowania. Informacje zamieszczone w tabeli zostały również uzupełnione (o znajdujące się w dalszej części rozdziału) przykłady i objaśnienia.

Tabela 2.1. *Symbole wyrażeń regularnych*

Symbol	Znaczenie
`.`	Dowolny znak.
`\`	Symbol poprzedzający znak specjalny. Na przykład ciąg `\.` oznacza znak kropki (`.`). Dodatkowo umieszczenie znaku `\` przed zwykłym znakiem może przekształcić go w symbol specjalny. Na przykład symbol `\t` odpowiada znakowi tabulacji.
`^`	Wyznacznik początku, oznaczający, że wzorzec rozpoczyna się wraz z początkiem ciągu tekstowego. Na przykład wzorzec `^A` oznacza, że ciąg tekstowy musi się rozpocząć literą `A`.
`$`	Wyznacznik końca, oznaczający, że ciąg tekstowy kończy się wraz z określonym wzorcem. Na przykład wzorzec `X$` oznacza, że ciąg tekstowy musi się zakończyć literą `X`.
`+`	Dopasowanie poprzedniego bloku jeden raz lub więcej razy. Na przykład wzorzec `a+` odpowiada jednej lub większej liczbie liter `a`.
`*`	Dopasowanie poprzedniego bloku zero lub więcej razy. Znaczenie symbolu jest takie samo, jak znaku `+`, z wyjątkiem faktu, że akceptowany jest również brak jakiejkolwiek wartości.
`?`	Dopasowanie poprzedniego bloku zero lub jeden raz. Innymi słowy, symbol ten definiuje poprzedzający go blok jako opcjonalny. Czyni również symbole `+` i `*` „niezachłannymi" (więcej informacji na temat zachłannego i niezachłannego dopasowywania znajduje się w kolejnym podrozdziale).
`()`	Funkcje grupowania i przechwytywania. *Grupowanie* oznacza przetwarzanie jednego lub większej liczby znaków w taki sposób, jakby były pojedynczym elementem. *Przechwytywanie* polega na zapamiętywaniu dopasowanego fragmentu, w celu późniejszego wykorzystania. Operacje te są nazywane wstecznym odwołaniem (ang. *backreference*).
`[]`	Symbole te wyznaczają *klasę znaków*. Za dopasowany uznaje się tylko jeden znak z wymienionego zbioru. Na przykład wzorzec `[abc]` odpowiada pojedynczemu znakowi o wartości `a`, `b` lub `c`.
`^`	Negacja dopasowania do zbioru znaków. Choć pozornie istnieje sprzeczność w interpretacji tego symbolu, w rzeczywistości nie ma tu żadnej niejednoznaczności. Znak `^` ma po prostu różne znaczenia w różnym kontekście. Na przykład wzorzec `[^abc]` odpowiada za dopasowanie pojedynczego znaku, który nie jest literą `a`, `b` ani `c`.
`!`	Umieszczenie symbolu na początku wyrażenia regularnego oznacza jego negację. Zatem akceptowane są ciągi, które nie są zgodne z podanym wzorcem[1].

[1] Składnia ta jest specyficzna dla wyrażeń regularnych modułu mod_rewrite i może nie być zgodna z wyrażeniami regularnymi stosowanymi w innych przypadkach.

Choć nie jest to pełna lista wyrażeń regularnych, stanowi dobry punkt wyjściowy do dalszej pracy. Znaczenie każdego wyrażenia regularnego przedstawionego w książce zostało wyjaśnione w dalszej części rozdziału. Opisy te zawierają przykłady, pozwalające na analizę praktycznego znaczenia symboli, prezentowanych w tabeli 2.1. Z mojego doświadczenia wynika, że wyrażenia regularne są znacznie łatwiejsze do zrozumienia dzięki analizie przykładów niż dzięki poznaniu teorii.

Szczegółowy opis poszczególnych pozycji tabeli 2.1 wraz z odpowiadającymi im przykładami został zamieszczony w kolejnych punktach podrozdziału.

Dopasowanie dowolnego znaku (.)

Znak kropki (`.`), umieszczony w definicji wyrażenia regularnego, odpowiada za dopasowanie dowolnego znaku. Jako przykład rozważmy wzorzec:

```
a.c
```

Za dopasowany zostanie uznany ciąg zawierający literę `a`, po której występuje dowolny znak i litera `c`. Zatem wzorzec ten odpowiada ciągom tekstowym `abc`, `asceta` i `marchew`. Każdy z nich zawiera bowiem zdefiniowane w wzorcu litery. Za zgodny nie zostanie natomiast uznany ciąg `palec`, ponieważ między literami `a` i `c` występują dwa znaki. Symbol kropki (`.`) odpowiada jedynie pojedynczemu znakowi.

Znaki specjalne (\)

Znak odwrotnego ukośnika dodaje specjalne znaczenie do kolejnego znaku lub je usuwa — zależnie o kontekstu. Na przykład wiadomo już, że znak kropki (`.`) ma specjalne znaczenie. Gdyby jednak konieczne było zdefiniowanie rzeczywistego znaku kropki w ciągu tekstowym, należałoby poprzedzić go znakiem odwrotnego ukośnika. Symbol `.` oznacza więc „dowolny znak”, natomiast symbol `\.` odpowiada faktycznemu znakowi kropki (`.`).

Niektóre znaki otrzymują specjalne znaczenie, gdy zostaną poprzedzone znakiem (`\`). Na przykład o ile znak `s` odpowiada rzeczywistej literze `s` w ciągu tekstowym, o tyle symbol `\s` zastępuje znak odstępu — znak spacji lub tabulacji.

Poprzedzenie znaku symbolem odwrotnego ukośnika nadaje mu specjalne znaczenie, przekształcając go w symbol specjalny. W dalszej części książki będą wykorzystywane również inne symbole specjalne, takie jak `\d` (znak cyfry) lub `\w` (znak „słowa”).

Określenie „symbol specjalny” często odnosi się do takich znaków, jak `.` i `$`, które mają szczególne znaczenie dla wyrażeń regularnych.

Wyznaczenie początku i końca ciągu (^ i $)

Znaki ^ i $ są nazywane **znakami kotwic**, ponieważ ich celem jest zapewnienie, że ciąg tekstowy rozpoczyna się od określonej sekwencji znaków i kończy określoną sekwencją znaków. Z uwagi na częstość występowania znaków kotwic są one wymienianie w grupie podstawowych symboli specjalnych.

Oto przykłady ich zastosowania:

```
^/
```

Powyższy wzorzec odpowiada dowolnemu ciągowi tekstowemu, który rozpoczyna się od znaku ukośnika.

```
\.jpg$
```

Ten wzorzec z kolei opisuje dowolny ciąg tekstowy, który kończy się sekwencją znaków `.jpg`.

```
^/$
```

Ostatni z przykładów wyznacza ciąg tekstowy, który rozpoczyna się i kończy znakiem ukośnika. Za dopasowany zostanie więc uznany jedynie ciąg złożony z jednego znaku ukośnika, bez jakichkolwiek dodatkowych znaków.

Dopasowanie jednego lub większej liczby znaków (+)

Symbol + pozwala na jednokrotne lub wielokrotne dopasowanie grupy znaków. Na przykład poniższy wzorzec zaakceptuje często popełniane błędy w zapisie angielskiego słowa *giraffe*.

```
giraf+e+
```

Umożliwia on bowiem wprowadzenie jednego lub większej liczby znaków `f` oraz jednego lub większej liczby znaków `e`. Zatem za dopasowane zostaną uznane ciągi `girafe`, `giraffe`, `giraffee`, a także `giraffffeeeee`.

Dopasowanie zera lub większej liczby znaków (*)

Symbol * pozwala na wystąpienie poprzedzającego go znaku zero lub więcej razy. Jego funkcja jest więc niemal identyczna z przeznaczeniem symbolu +. Jedyna różnica polega na tym, że akceptowane są również ciągi, które nie zawierają w ogóle sekwencji zdefiniowanych we wzorcu. W praktyce symbol ten jest często wykorzystywany w przypadkach, w których powinien zostać zastosowany symbol +. Problemem jest wówczas fakt, że za dopasowany zostaje uznany również pusty ciąg tekstowy. Przykładem może tu być nieznacznie zmodyfikowany wzorzec z poprzedniego punktu.

```
giraf*e*
```

Będzie on odpowiadał ciągom wymienionym poprzednio (giraffe, girafe i giraffee), ale będzie również zgodny z ciągiem giraeeee, który nie zawiera żadnych znaków f, czy z ciągiem gira, w którym nie występują ani znaki f, ani e.

Najczęściej symbol ten wykorzystuje się w połączeniu z symbolem ., oznaczającym dopasowanie dowolnej treści. Często osoby stosujące je zapominają, że wyrażenia regularne dopasowują podciągi. Jako przykład warto tu rozważyć wzorzec:

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

Z założenia wzorzec tego typu powinien akceptować ciągi tekstowe kończące się sekwencją .gif. Takie zakończenie jest wymuszane przez symbol $. Z kolei symbol \ poprzedzający znak . sprawia, że znak kropki jest traktowany jako rzeczywisty znak, a nie symbol specjalny. W tym przypadku celem zastosowania sekwencji .* było wskazanie, że ciąg może się rozpoczynać od dowolnych znaków. W rzeczywistości jest ona jednak zbędna i niepotrzebnie zajmuje czas procesora podczas wykonywania zadania.

Bardziej użytecznym przykładem wykorzystania symbolu * jest zadanie wyszukania wierszy komentarzy w pliku konfiguracyjnym serwera Apache. W takim przypadku pierwszym znakiem (innym niż znak odstępu) musi być #. Niemniej trzeba również uwzględnić możliwość poprzedzenia go znakami spacji.

```
^\s*#
```

Wzorzec ten odpowiada za dopasowanie ciągu tekstowego; może on (ale nie musi) rozpoczynać się znakami odstępu, po których występuje znak #. Wymusza więc stosowanie znaku #, jako pierwszego znaku (poza znakami odstępu) w wierszu.

Zachłanne dopasowywanie

W przypadku stosowania symboli + i * mamy do czynienia z **zachłannym** dopasowywaniem. Wyrażenie regularne dopasowuje wówczas możliwie największą liczbę znaków. Oznacza to, że zastosowanie wyrażenia regularnego a+ w odniesieniu do ciągu aaaa spowoduje, że za dopasowane zostaną uznane wszystkie znaki, a nie tylko pierwsza litera a. Ma to szczególne znaczenie w przypadku wyrażenia .*, które niekiedy może objąć większą część ciągu, niż powinno. Przykład tego problemu zostanie przedstawiony po omówieniu kilku kolejnych symboli specjalnych.

Dopasowanie opcjonalne (?)

Symbol ? wyznacza opcjonalny pojedynczy znak. Jest on niezwykle użyteczny, gdy trzeba uwzględnić często występujące błędy w pisowni lub elementy, które mogą (ale nie muszą) być zapisane w ciągu. Na przykład można go wykorzystać w słowie, co do którego nie mamy pewności, czy powinno zawierać znak myślnika, czy nie.

```
e-?mail
```

Podany wzorzec akceptuje zarówno ciąg email, jak i e-mail, więc zwolennicy obydwu form zapisu będą usatysfakcjonowani.

Dodatkowo znak ? wyłącza „zachłanność" symboli + i *. Zatem umieszczenie znaku ? za symbolami + i * sprawia, że wyrażenie dopasuje możliwie najmniej znaków (zobacz wcześniejsze informacje na temat zachłannego dopasowania).

Na przykład, jeśli wzorzec `g.*a` zostanie zastosowany w odniesieniu do ciągu `gramatyka`, symbole `.*` będą odpowiadały za dopasowanie fragmentu `ramatyk`. Gdyby jednak wzorzec miał postać `g.*?a`, wyrażenie `.*` nie miałoby charakteru zachłannego i odpowiadałoby za dopasowanie jedynie pierwszej litery `r`.

Kolejne porównania zachłannych i niezachłannych wyrażeń zostały zamieszczone w części występującej za omówieniem odwołań wstecznych.

Grupowanie i przechwytywanie (())

Znaki nawiasów umożliwiają grupowanie kilku znaków w jeden element oraz przechwytywanie wyników dopasowania w celu późniejszego wykorzystania. Możliwość przetwarzania kilku znaków jako jednego elementu jest niezwykle użyteczna w przypadku dopasowywania treści do wzorca. Przedstawiony poniżej przykład, choć niezbyt przydatny, jest poprawny pod względem funkcjonalnym.

```
(abc)+
```

Odpowiada on za wyszukanie sekwencji `abc`, występującej jeden lub więcej razy. Za dopasowany zostanie więc uznany ciąg `abc`, jak również ciąg `abcabc`.

Jeszcze bardziej użyteczną funkcją znaków nawiasu jest zdolność „przechwytywania" wyniku. Gdy ciąg zostanie dopasowany do wzorca, często zachodzi konieczność zapamiętania jego treści, by można ją było wykorzystać w dalszych operacjach. Takie późniejsze wykorzystanie nazywa się **odwołaniem wstecznym**.

Może na przykład wystąpić konieczność wyszukania pliku *.gif* (podobnie jak we wcześniejszym przykładzie), ale w sposób, który pozwoli na określenie, jaka w rzeczywistości była nazwa dopasowanego pliku. Dzięki zastosowaniu znaków nawiasu przechwycona nazwa może zostać wykorzystana w późniejszych operacjach przetwarzania.

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

W przypadku dopasowania treści do wzorca, przechwycona wartość zostanie zapisana w specjalnej zmiennej `$1` (w niektórych przypadkach nazwą zmiennej może być `%1`[2]). Jeżeli w wyrażeniu zostało zdefiniowanych więcej nawiasów, druga wartość zostanie przechwycona w zmiennej `$2`, trzecia w `$3` itd. Ostatnią z możliwych do wykorzystania zmiennych jest `$9`. Ograniczenie wynika z faktu, że zmienna `$10` byłaby niejednoznaczna. Mogłaby odpowiadać zmiennej `$1`, po której występuje rzeczywisty znak zera (`0`) lub zmiennej `$10`. Zamiast wprowadzać dodatkowe symbole specjalne wyróżniające poszczególne przypadki, programiści modułu mod_rewrite zdecydowali się na ograniczenie zmiennych odwołania wstecznego do `$9`.

[2] W regułach `RewriteRule` zmienne są poprzedzane znakiem dolara (`$`), natomiast w regułach `RewriteCond` są zapisywane za znakiem wartości procentowej (`%`). Więcej informacji na temat dyrektyw `RewriteRule` i `RewriteCond` znajduje się w rozdziałach 4. i 5.

Praktyczny sposób wykorzystania tej opcji zostanie przedstawiony w rozdziale 3., dotyczącym dyrektywy `RewriteRule`.

Analizując ponownie przykład opisujący dopasowanie zachłanne i niezachłanne, warto rozważyć dwa następujące wzorce, zastosowane w odniesieniu do ciągu `gramatyka`.

```
g(.*)a
g(.*?)a
```

Pierwszy wzorzec zwróci w zmiennej `$1` wartość `ramatyk`, natomiast drugi spowoduje zapisanie w zmiennej `$1` wartości `r`. Gdy wyrażenie jest analizowane w trybie zachłannym, dopasowuje jak najwięcej znaków i zatrzymuje swoje działanie po odczytaniu ostatniego znaku `a`. W trybie niezachłannym zadanie zostanie zrealizowane po odczytaniu najmniejszej wystarczającej liczby znaków, czyli po odczytaniu pierwszej litery `a`.

Aby mieć pewność, w jaki sposób będą dopasowywane do wzorców poszczególne fragmenty ciągów, warto wykorzystać takie narzędzia programowe, jak np. Regex Coach. Szczegółowy opis odwołań wstecznych, zachłannego dopasowywania i procesów zachodzących w trakcie dopasowywania został zamieszczony w książce pt. *Wyrażenia regularne* (Helion, 2001).

Dopasowanie jednego znaku ze zbioru ([ ])

Znaki klasy (`[]`) umożliwiają definiowanie zbioru znaków, z których każdy spełnia kryteria dopasowania. Wyrażenia regularne udostępniają kilka wbudowanych klas znaków (definiowanych na przykład za pomocą symbolu specjalnego `\s`, prezentowanego wcześniej), ale pozwalają również na wyznaczanie własnych klas. Oto bardzo prosty przykład opisywanego mechanizmu:

```
[abc]
```

Zdefiniowana klasa znaków odpowiada za dopasowanie litery `a`, `b` lub `c`. Na przykład, gdyby było konieczne ustalenie podzbioru użytkowników, których nazwy rozpoczynają się od jednej z wymienionych liter, można by zastosować następujący wzorzec dopasowania:

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

Wzorzec ten zawiera kilka znaków, które zostały wcześniej opisane. Jego zadanie polega na dopasowaniu porównywanego ciągu do ścieżek dostępu do katalogów pewnego podzbioru użytkowników i zapisaniu nazw poszczególnych użytkowników w zmiennej `$1`. Choć, jak się za chwilę okaże, nie do końca jest to prawda.

W definicji klasy znaków można również dość łatwo wyznaczać zakres akceptowanych wartości. Na przykład, gdyby było konieczne dopasowanie wartości liczbowych z przedziału od `1` do `5`, należałoby zastosować definicję klasy o treści `[1-5]`.

Jeżeli w definicji klasy znaków jako pierwszy występuje symbol ^, otrzymuje on specjalne znaczenie. Klasa wyznaczona za pomocą ciągu `[^abc]` jest przeciwieństwem klasy `[abc]`. Odpowiada więc każdemu znakowi, który nie jest znakiem `a`, `b` lub `c`.

W tym miejscu warto powrócić do wcześniejszego przykładu — próby dopasowania nazw użytkowników rozpoczynających się od liter `a`, `b` i `c`. Problem w tym, że w prezentowanym rozwiązaniu znak * ma charakter zachłanny, czyli dopasowuje możliwie najwięcej znaków. Aby zatrzymać dopasowywanie po odczytaniu znaku ukośnika, konieczne byłoby zdefiniowanie wzorca, który akceptowałby wszystkie znaki oprócz znaku ukośnika.

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

Sekwencja `.*` została zastąpiona wyrażeniem `[^/]+`, które w praktyce oznacza dopasowanie dowolnych znaków poprzedzających znak ukośnika lub koniec ciągu tekstowego (zależnie od tego, co będzie pierwsze). Dodatkowo zamiast symbolu * został użyty symbol + — nazwa użytkownika składająca się z jednego znaku zazwyczaj nie jest akceptowana. Po tej operacji w zmiennej `$1` jest zapisana nazwa użytkownika, a nie nazwa użytkownika i ewentualne elementy ścieżki dostępu do innego katalogu (występujące za nazwą).

Negacja (!)

Jeżeli zachodzi konieczność zanegowania całego wyrażenia regularnego, wystarczy poprzedzić je znakiem `!`. Rozwiązanie to nie jest uniwersalne dla wszystkich implementacji wyrażeń regularnych, ale może być wykorzystywane w znacznej większości z nich. W przypadku reguł przepisywania bardzo często jest ono stosowane do zaznaczenia, że dany wzorzec odnosi się do wszystkich katalogów, oprócz jednego. Zatem aby wyłączyć ze zbioru przetwarzanych katalogów katalog */obrazy*, należy wyznaczyć wyrażenie dopasowania do katalogu */obrazy*, a następnie je zanegować.

```
!^/obrazy
```

Taka definicja pozwala na zaakceptowanie wszystkich ścieżek dostępu, które nie rozpoczynają się od ciągu `/obrazy`. Więcej podobnych przykładów znajduje się w dalszej części książki, a szczególnie w rozdziale 11.

Przykłady wyrażeń regularnych

W zrozumieniu zasad działania wyrażeń regularnych pomocnych może być kilka przykładów. Opis poszczególnych wyrażeń regularnych rozpoczyna się od często spotykanych przypadków, które zostały uzupełnione o pewne rozwiązania alternatywne.

Adresy poczty elektronicznej

Prezentację przykładów rozpoczyna wyrażenie rozwiązujące często występujący problem — przygotowanie wzorca, który pozwoli na sprawdzenie adresu poczty elektronicznej[3]. Ogólna składnia adresu e-mail to: `coś@coś.coś`. Przygotowując wyrażenie regularne od podstaw, warto zawsze opisać sobie wzorzec właśnie w takiej formie, gdyż jest to dobry punkt wyjścia do prac nad samą treścią wyrażenia.

Aby opisać adres e-mail za pomocą wyrażenia regularnego, trzeba przeanalizować jego poszczególne elementy składowe. Fragmenty zapisane jako `coś` można łatwo wyrazić za pomocą sekwencji `.+`. Natomiast fragment `@` jest rzeczywistym znakiem, występującym w treści adresu.

Uzyskujemy więc wzorzec o następującej treści:

```
.+@.+\..+
```

Jest to dobry punkt wyjściowy, odpowiadający większości adresów poczty elektronicznej; prawdopodobnie nawet wszystkim. Jednak za dopasowane zostaną uznane również inne ciągi, które nie są adresami e-mail (np. `@@@.@` czy `@.com`). Trzeba się więc zastanowić nad bardziej szczegółową weryfikacją.

W pierwszej kolejności należałoby się upewnić, że ciąg występujący przed znakiem `@` nie jest pusty i składa się ze znaków określonego rodzaju. Powinny to być znaki alfanumeryczne i niektóre znaki specjalne (takie jak znaki kropki, podkreślenia czy myślnika).

Na szczęście wyrażenia zgodne ze standardem PCRE udostępniają wygodne mechanizmy wskazywania „znaków alfanumerycznych" — za pomocą predefiniowanej klasy znaków. Istnieje klika wstępnie przygotowanych klas tego typu, między innymi `[:alpha:]` (dopasowanie liter), `[:digit:]` (dopasowanie cyfr od 0 do 9) i `[:alnum:]` (dopasowanie znaków alfanumerycznych).

Kolejna czynność powinna polegać na wymuszeniu stosowania znaków alfanumerycznych w części nazwy domeny, ale z pominięciem domeny najwyższego poziomu (ostatniego elementu nazwy domenowej), która może zawierać jedynie litery. Dawniej podczas wyznaczania wzorców można było ograniczyć tę część do trzech liter. Jednak obecnie wykorzystywane są domeny nie spełniające tego założenia.

Ostatnim zadaniem jest umożliwienie wykorzystania dowolnej liczby znaków kropki w nazwie serwera, tak aby zarówno ciąg `a.com`, jak i `poczta.wsg.byd.pl` były uznawane za poprawne elementy składowe adresu e-mail.

Po przekształceniu wymienionych założeń do postaci wyrażenia otrzymujemy następujący ciąg:

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

[3] Nie jest to wzorzec występujący szczególnie często w dyrektywach modułu mod_rewrite, lecz w definicjach wyrażeń regularnych jako takich. Adresy poczty elektronicznej rzadko występują w ciągach URL, dlatego sporadycznie są opisywane przez dyrektywy mod_rewrite.

Taka definicja jest z pewnością bardziej szczegółowa i zapewni poprawną weryfikację adresu poczty elektronicznej. Oczywiście nadal istnieje możliwość dopasowania ciągu nie odpowiadającego adresowi e-mail, ale prawdopodobieństwo takiego zdarzenia jest niewielkie.

Numer telefoniczny

Kolejnym rozważanym zagadnieniem jest sprawdzenie numeru telefonicznego. Zdefiniowanie wzorca jest znacznie trudniejsze, niż można by sądzić na początku. Aby ułatwić sobie zadanie, ograniczymy się jedynie do numerów stosowanych w Polsce, składających się z dziesięciu cyfr.

Numer telefoniczny stanowią trzy cyfry numeru kierunkowego i siedem cyfr abonenta. Cyfry te mogą, ale nie muszą, być rozdzielane za pomocą różnych znaków. Pierwsze trzy z nich są niekiedy otoczone znakami nawiasu. Rozwiązaniem może być następujący wzorzec:

```
\(?\d{3}\)?[- ]?\d{3}[- ]?\d{2}[- ]?\d{2}
```

Odpowiada on większości polskich numerów telefonicznych, zapisywanych w typowych formatach. Trzy pierwsze cyfry mogą, ale nie muszą, być umieszczone w nawiasie, a bloki trzech, dwóch i dwóch kolejnych cyfr mogą być rozdzielane za pomocą spacji lub znaków myślnika, bądź pozostać nie rozdzielone w ogóle. Zaproponowane rozwiązanie jest dalekie od doskonałości, ponieważ użytkownicy znajdą sposób na wprowadzenie danych w formacie innym niż założony.

Przeanalizujmy regułę symbol po symbolu.

Symbol `\(?` reprezentuje opcjonalny znak otwarcia nawiasu. Znak odwrotnego ukośnika jest niezbędny, ponieważ nawias ma szczególne znaczenie, opisane we wcześniejszej części rozdziału. W tym wyrażeniu uwzględniony jest rzeczywisty znak nawiasu, a nie symbol specjalny. Znak zapytania z kolei przekształca znak nawiasu w element opcjonalny. Oznacza to, że osoba wprowadzająca dane może, ale nie musi, zapisać trzy pierwsze cyfry w nawiasie. Każda z metod jest akceptowana.

Symbol `\d{3}` obejmuje dwa elementy, które nie były dotychczas opisywane. Sekwencja `\d` oznacza cyfry. Ta sama definicja może być zapisana za pomocą elementu `[:digit:]`, ale notacja `\d` jest znacznie częściej stosowana i wymaga mniej pisania. Ciąg `{3}` umieszczony za ciągiem `\d` wskazuje, że dany znak musi wystąpić trzy razy. W tym przypadku oznacza to, że konieczne jest wprowadzenie trzech cyfr. W przeciwnym razie dopasowanie zakończy się błędem.

Notacja typu `{n}` występuje jeszcze w dwóch formach składniowych, zależnych od liczby powtórzeń znaków. Wspomniane formy zapisu zostały zestawione w tabeli 2.2.

Tabela 2.2. *Składnia wzorca powtórzenia {n,m}*

Składnia	Znaczenie
`{n}`	Znak musi zostać wymieniony dokładnie `n` razy.
`{n,}`	Znak musi zostać wymieniony `n` razy, ale dopuszczalne są kolejne powtórzenia.
`{n,m}`	Znak musi zostać wymieniony przynajmniej `n` razy, ale nie więcej niż `m` razy.

Symbol `\)?` oznacza opcjonalny znak zamknięcia nawiasu — analogicznie do znaku otwierającego nawias.

Sekwencja `[- ]?` definiuje kolejny znak opcjonalny. Jest to znak myślnika lub spacji (bądź jego brak), rozdzielający pierwsze trzy cyfry od kolejnych trzech cyfr.

Pozostała część wyrażenia zawiera opisane już elementy. Jedyna różnica polega na liczbie cyfr, które są wymagane w poszczególnych blokach.

Kolejnym etapem przygotowywania wyrażenia regularnego powinno być opracowanie sposobów „obejścia" zabezpieczenia i ustalenie, czy obsługa tych skrajnych przypadków jest warta dodatkowej pracy. Na przykład niektórzy użytkownicy będą poprzedzali numer znakami `+48`. W niektórych numerach dopisywane będą na końcu numery wewnętrzne. Z pewnością również znajdzie się osoba, która będzie chciała rozdzielać bloki cyfr za pomocą innego znaku niż zdefiniowane. Problem oczywiście można rozwiązać za pomocą bardziej złożonego wyrażenia regularnego. Jednak wzrost złożoności wiąże się z obniżeniem szybkości działania i zmniejszeniem czytelności dyrektywy. Opis przedstawionego wyrażenia zajął całą stronę, co najlepiej oddaje problem czasu, który będzie potrzebny na rozszyfrowanie wyrażenia przy ponownej jego analizie po kilku miesiącach od utworzenia (gdy jego autor zapomni już, do czego ono służyło).

Dopasowanie ciągów URI

Ponieważ książka ta jest poświęcona modułowi mod_rewrite, warto przeanalizować kilka przykładów dopasowania ciągów URI, gdyż to one będą podstawą wszelkich działań opisywanych w dalszej części publikacji.

Większość dyrektyw, wymienionych w pozostałej części książki, wymaga podania wyrażenia regularnego jako jednego z parametrów. W większości przypadków wyrażenia te będą opisywały ciąg URI, który jest techniczną nazwą dla zasobu wymienionego w żądaniu, przekazanym do serwera. Zazwyczaj odpowiada on treści zamieszczonej po adresie *http://www.domena.pl*.

W kolejnych punktach zostaną opisane typowe przykłady wartości, które podlegają dopasowaniu za pomocą wyrażeń regularnych.

Dopasowanie strony głównej

Niekiedy zachodzi konieczność dopasowania strony głównej danej witryny. Zazwyczaj właściwym dla tego przypadku adresem URI jest ciąg pusty, ciąg złożony ze znaku / lub z nazwy strony indeksu (np. */index.html* lub */index.php*). Pusty ciąg URI występuje wówczas, gdy zapisany w żądaniu adres nie zawiera końcowego znaku ukośnika (np. *http://www.przyklad.pl*).

W pierwszej kolejności rozważmy przypadek, w którym przesłane przez użytkownika żądanie zawiera adres *http://www.przyklad.pl* lub *http://www.przyklad.pl/* (tj. ciąg adresu z końcowym znakiem ukośnika lub bez niego, ale bez określonego pliku strony). Celem wyrażenia jest więc dopasowanie opcjonalnego znaku ukośnika.

Zgodnie z informacjami przedstawionymi wcześniej, znaki opcjonalne są wyznaczane za pomocą symbolu `?`. Zatem wyrażenie powinno mieć postać:

```
^/?$
```

Odpowiada ono za dopasowanie ciągów tekstowych, które rozpoczynają się opcjonalnym znakiem ukośnika i nim kończą. Innymi słowy, akceptuje jedynie ciągi tekstowe, które rozpoczynają się i kończą znakiem ukośnika lub rozpoczynają się i kończą „niczym".

W kolejnym kroku wyrażenie powinno zostać uzupełnione o bardziej złożone nazwy plików. Za dopasowane powinny być uznawane cztery następujące ciągi tekstowe:

- Pusty ciąg (gdy użytkownik zapisze adres *http://www.przyklad.pl* bez końcowego ukośnika).
- Ciąg / (gdy użytkownik zapisze adres *http://www.przyklad.pl/* z końcowym ukośnikiem).
- Ciąg `/index.html`.
- Ciąg `/index.php`.

Podstawą dla wyrażenia będzie reguła opracowana poprzednio.

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

Nie jest to całkiem poprawna postać (o czym się będzie można przekonać, czytając dalszą część rozdziału), ale jest bliska ideałowi. Zawiera natomiast element, który nie był dotychczas opisywany — symbol `|`, reprezentujący alternatywę, czyli możliwość wyboru jednej lub drugiej opcji. W przypadku ciągu `(html|php)` oznacza to spełnienie kryteriów dla rozszerzenia `html` lub `php`.

Wyrażenie definiuje więc ciąg tekstowy, który rozpoczyna się od opcjonalnego znaku ukośnika, po którym występuje ciąg `index.`, a za nim ciąg `html` lub `php`. Cały ciąg (począwszy od słowa `index`) jest opcjonalny, a po nim nie występuje żadna wartość.

Jedyna wada wyrażenia polega na tym, że uznaje za dopasowane również ciągi `index.php` i `index.html` pozbawione poprzedzających je znaków ukośnika. Choć jest to sytuacja niedozwolona w tym kontekście wykorzystania ciągu URI, w praktyce nie powinna być powodem szczególnego zmartwienia. Mimo że użytkownik mógłby przesłać żądanie w jednej z opisanych postaci, można przyjąć, że w najprawdopodobniej tego nie zrobi, a nawet gdyby tak uczynił, to nic się nie stanie, jeżeli żądanie zostanie potraktowane w taki sam sposób, jakby zawierało poprawną wartość URI.

Dopasowanie katalogu

Gdyby konieczne było ustalenie, jaki katalog został zapisany w ciągu URI lub jakim słowem kluczowym się rozpoczyna, trzeba byłoby wyodrębnić całą treść, występującą przed pierwszym znakiem ukośnika. Wyrażenie miałoby wówczas następującą postać:

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

Składa się ono z kilku elementów. Pierwszym jest standardowy, często stosowany, symbol `^/`, oznaczający „rozpoczęcie od znaku ukośnika". Kolejny wyznacza klasę znaków `[^/]`, która odpowiada za dopasowanie wszystkich znaków oprócz ukośnika. Symbol `+` oznacza natomiast, że liczba tych znaków powinna wynosić jeden lub więcej. Z kolei nawias zapewnia zapisanie wartości w zmiennej `$1` do późniejszego wykorzystania.

Dopasowanie rozszerzenia pliku

Trzecim z prezentowanych przykładów jest rozwiązanie umożliwiające dopasowanie plików o określonym rozszerzeniu — również bardzo często wykorzystywane. Załóżmy, że celem wzorca jest wyodrębnienie plików graficznych. Wyrażenie regularne przedstawione poniżej znajduje zastosowanie w odniesieniu do większości najczęściej wykorzystywanych rodzajów plików graficznych:

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

W dalszej części książki zostaną również przedstawione sposoby ignorowania wielkości znaków, co umożliwi dopasowywanie również rozszerzeń zapisanych wielkimi literami.

Narzędzia wspomagające pracę z wyrażeniami regularnymi

Osoby, które zajmują się wyrażeniami regularnymi przez dłuższy czas, prędzej czy później sięgają po narzędzia programowe, zapewniające wsparcie i wizualizację realizowanych procesów. Programów wspomagających pracę jest wiele, a każdy z nich ma pewne wady i zalety. Nie ulega wątpliwości, że większość najbardziej użytecznych aplikacji opracowywania wyrażeń regularnych jest dostarczanych przez programistów skupionych wokół projektu Perl. Wynika to z faktu, że wyrażenia regularne są szczególnie często stosowane w języku Perl i występują w niemal każdym programie.

Aplikacja Regex Coach

Jedną z wartych polecenia aplikacji jest Regex Coach. Program ten jest dostępny zarówno dla platformy Linux, jak i Windows, i można go pobrać ze strony *http://www.weitz.de/regex-coach/*. Umożliwia on krokowe wykonanie wyrażenia regularnego i przeglądanie dopasowanych fragmentów ciągu. Funkcja ta jest niezwykle użyteczna, gdy użytkownik uczy się projektować własne wyrażenia.

Podsumowanie

Zapoznanie się z podstawowymi informacjami na temat wyrażeń regularnych jest niezbędne przed przystąpieniem do pracy z modułem mod_rewrite. Zbyt często administratorzy starają się opracowywać wyrażenia stosując metody siłowe — wprowadzają różne kombinacje parametrów do chwili, gdy opracowany wzorzec wydaje się spełniać swoje zadanie. Rezultatem takich działań są nieefektywne i niestabilne wyrażenia, których przygotowanie zabiera wiele czasu i wywołuje frustracje.

Warto włożyć zakładkę w tę część książki i wracać do lektury rozdziału za każdym razem, gdy konieczne będzie ustalenie, jakie funkcje realizuje dane wyrażenie.

Do innych źródeł informacji, wartych polecenia, można zaliczyć dokumentację wyrażeń regularnych języka Perl, dostępną pod adresem *http://perldoc.perl.org/perlre.html*) i po wpisaniu polecenia `perldoc perlre`, a także dokumentację PCRE, która jest dostępna pod adresem *http://pcre.org/pcre.txt*.