Doing Web Apps

Reguläre Ausdrücke

Autor: Rüdiger Marwein
Letzte Änderung: 2009-04-17
Version: 0.6
Copyright: 2005. Alle Rechte vorbehalten

Dieses Dokument darf – mit Nennung des Autoren - frei vervielfältigt, verändert und weitergegeben werden.

Der Inhalt ist sorgfältig recherchiert, mit dem Dokument ist jedoch keinerlei Garantie auf Fehlerfreiheit gewährleistet.

Dieser Inhalt ist unter einem Creative Commons Namensnennung Lizenzvertrag lizenziert. Um die Lizenz anzusehen, gehen Sie bitte zu http://creativecommons.org/licenses/by/2.0/de/ oder schicken Sie einen Brief an Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Inhaltsverzeichnis

1.Einleitung....3
2.Aufbau....3
3.Beispiele....3
3.1.preg_match....3
3.2.preg_replace....5
4.Regex sind cool....5
5.Allgemeine Hinweise....6
5.1.str_replace....6
5.2.strpos....7
5.3.substr....7
6.Quick Reference....8
6.1.Steuerzeichen....8
6.2.Zeichen-Gruppen....8
6.3.Ersetzungen....9
6.4.Wiederholungen....9
6.5.Optionen....10
6.6.Erweiterter Regulärer Ausdruck....10
6.7.Gruppierung....10

1.Einleitung

Reguläre Ausdrücke (oder engl.: Regular Expressions) sind so etwas wie erweiterte Suchbegriffe. Man legt nicht nur fest, ob man eine bestimmte Zeichenkette sucht, sondern man kann ebenso definieren, dass der Suchbegriff einem Schema / Muster entsprechen muss. Z.B. dass eine Datum das Format DD.MM.YYYY (also z.B. 22.07.1979) haben soll.

Um solche Aufgaben im kleinen zu erledigen ohne Algorithmen zum Parsen des Schemas entwickeln zu müssen, kann man reguläre Ausdrücke formulieren, die mit relativ wenig Aufwand für den Programmierer diese Leistung erbringen.

Wir behandeln hier die **Perl Regular Expressions** weil sie die meist verbreiteten und am weitesten entwickelten sind.

2.Aufbau

Ein Regulärer Ausdruck hat die Form

```
/<Suchbegriff bzw Ausdruck>/<optionen>
```

Also einen Suchbegriff (**Pattern**) eingeschlossen in Schrägstriche[1]. Dahinter können Optionen angegeben werden (z.B. dass die Gross-Kleinschreibung ignoriert werden soll).

Anhand einiger Beispiele soll schrittweise das Verständnis für immer komplexer werdende Ausdrücke geschaffen werden.

3.Beispiele

Mit PHP lassen sich reguläre Ausdrücke mit den Funktionen

```
preg_match
preg_match_all
preg_replace
```

behandeln.

preg_match liefert true, wenn der Suchbegriff (**Pattern**) gefunden wurde. Es kann die gefundenen Zeichenketten (**Matches**) in eine separat angegebene Variable zurückschreiben, falls nötig.

preg_match_all vollzieht die Suche über das komplette Dokument (String) und hört nicht beim ersten Treffer auf.

preg_replace ersetzt einen String, der einem regulärem Ausdruck entspricht durch eine Zeichenkette. Bei der Ersetzung können auch Teile des Treffers verwendet werden.

3.1.preg_match

Problem: Eine Zeichenkette muss das Wort „spielen“ beinhalten.

1 Schrägstriche sind in anderen Sprachen üblich. Bei PHP kann es ein beliebiges Zeichen sein, welches den Ausdruck umschließt. Sollte das Zeichen im Pattern vorkommen muss es mit einem \ (Backslash) escaped werden.

Lösung: `if( preg_match('/spielen/',$string) ) echo 'OK';`

Erklärung: Ein Wort, das keine reservierten Zeichen beinhaltet, kann einfach so eingesetzt werden.

Problem: Eine Zeichenkette muss das Wort „spielen“ oder „basteln“ beinhalten

Lösung: `if( preg_match('/spielen|basteln/',$string) ) echo 'OK';`

Erklärung: Der Hoch-Strich (**Pipe**) ist der Oder-Trenner. | ist ein reserviertes Zeichen.

Problem: Eine Zeichenkette muss das Wort „spielen“ oder „basteln“ beinhalten, wobei der erste Buchstabe gross oder klein sein kann.

Lösung: `if( preg_match('/[sS]pielen|[bB]asteln/',$string) ) echo 'OK';`

Erklärung: In [] eingeschlossene Zeichen werden als Gruppe von Zeichen, die an eben dieser Stelle vorkommen dürfen, interpretiert. Es können auch Steuerzeichen eingesetzt werden. Reservierte Zeichen müssen mit einem Backslash versehen werden. **Die Gruppe steht für genau ein Zeichen**.

Problem: Eine Zeichenkette muss das Wort „spielen“ oder „basteln“ beinhalten, wobei die Gross-Kleinschreibung völlig egal ist.

Lösung: `if( preg_match('/spielen|basteln/i',$string) ) echo 'OK';`

Erklärung: Die Option „i“ am Ende des Ausdrucks heisst „case-insensitive“, also „nicht Gross-Kleinschreibungs-empfindlich“.

Problem: Es soll geprüft werden, ob eine Zeichenkette eine URL beinhaltet.

Lösung: `if( preg_match('/http:\/\/.+?\ /',$string) ) echo 'OK';`

Erklärung: Eine URL beginnt hier per Definition mit „http://“. Da der Schrägstrich ein reserviertes Zeichen ist muss er **gequotet** (gebackslashed / entwertet / escaped) werden. Der **Punkt** steht für ein **beliebiges** Zeichen. Das **Plus** besagt, dass das davor angegebene beliebige Zeichen **1-n mal** vorkommen darf bis ein Leerzeichen auftaucht. Dies wird durch das eingebettete **Fragezeichen** möglich. Das Leerzeichen muss auch escaped werden.

Problem: Vom Benutzer wurde eine E-Mail angegeben und nun soll geprüft werden, ob diese theoretisch korrekt ist.

Lösung: `if( preg_match('/^[a-z0-9\.\_]+?@[a-z\.\_]*/i', $string) ) echo 'OK';`

Erklärung: Das ^ (**Dach**) besagt, dass das nachfolgende Zeichen am **Beginn der Zeichenkette** stehen muss. Vor dem @-Zeichen dürfen Buchstaben, Zahlen, Punkte und Unterstriche stehen. Das Plus besagt, dass vor dem @ mindestens eins dieser Zeichen stehen muss. Nach dem @ muss es mit diesen Zeichen weiter gehen. Diese Abfrage ist bei weitem nicht perfekt gelöst und durchaus ausbaufähig. Das **Sternchen** besagt, dass **0-n mal** ein Zeichen der zuvor genannten Zeichengruppe folgen können.

Problem: Der Benutzer musste ein Datum eingeben und es muss das Format DD-MM-YYYY haben.

Lösung: `if( preg_match('/^[0-9]{2}\-[0-9]{2}\-[0-9]{4}$/', $string) ) echo 'OK';`

Erklärung: Die Zahl in geschweiften Klammer besagt wie oft das Zeichen vorkommen muss. {2,} würde bedeuten: mindestens zwei mal. Der Punkt ist ein Steuerzeichen und wird escaped. ^ (Dach) bezeichnet den Stringanfang, $ (**Dollar**) bezeichnet das Stringende.

Problem: In einer Zeichenkette dürfen die Buchstaben ä,ö und ü nicht vorkommen.

Lösung: `if( preg_match('/[^äöü]/', $string) ) echo 'OK';`

Erklärung: In den []-Klammer bedeutet das ^ (Dach) die Negation. Also eben dass kein ä,ö und ü stehen darf.

3.2.preg_replace

Problem: Alle Gänsefüßchen sollen durch das HTML-Äquivalent " ersetzt werden.

Lösung: $string = preg_replace('/"/','"',$string);

Problem: Schreibfehler müssen korrigiert werden. Überall wo „meinefunktion“ oder „meineFunktion“ steht, soll eigentlich MeineFunktion stehen.

Lösung: `$string = preg_replace('/meine[fF]unktion/', 'MeineFunktion',$string);`

Problem: Ein Datum der Form DD.MM.YYYY soll umgewandelt werden in ein Datum der Form YYYY-MM-DD.

Lösung: `$string = preg_replace('/^([0-9]{2})\.([0-9]{2})\.([0-9]{4})$/','\\3-\\2-\\1',$string));`

Erklärung: Die Teile, die man wiederverwenden will, werden geklammert. Beim Ergebnis ist der Nullte Treffer das gesamte Konstrukt. Der erste Treffer ist das erste von links mit einer Klammer geöffnete Teil-Konstrukt. Die geklammerten Treffer können im Ersetzungsstring mit zwei Backslashes eingesetzt werden.

Problem: Eine Zeichenkette muss in eine URL-konforme Schreibweise transformiert werden. Aus „Über meine (größten) 2 Errungenschaften“ soll am Ende so aussehen: „ueber_meine_groessten_2_errungenschaften“.

Lösung:

```
// kleinbuchstaben erzwingen und umlaute ersetzen
$string = strtr(
  strtolower($string),
  array('ä'=>'ae','ö'=>'oe','ü'=>'ue','ß'=>'ss')
);
$string = preg_replace('/[^a-z0-9]/',' ',$string));
$string = preg_replace('/[ ]+/','_',$string));
```

Erklärung: Zwei Vorbedingungen – Kleinbuchstaben mit ersetzten Umlauten – lassen sich in der Vorverarbeitung ohne einen regulären Ausdruck erledigen.

Im übrig gebliebenen Konstrukt wird nun alles, was keine Zahl ist zu einem Leerzeichen umgewandelt. Es können nun mehrere Leerzeichen in Folge entstanden sein. Diese Leerzeichen(-folgen) werden am Ende noch je durch einen Unterstrich ersetzt.

4.Regex sind cool

Wo braucht man diese komische Konstrukte jetzt eigentlich außer ab und zu in PHP?

Prinzipiell kommen sie ja von Perl, also braucht man sie dort definitiv.

Dort verwendet man sie so:

```
if ( $text=~/meine[fF]unktion/ )
    print "Schreibfehler entdeckt";
```

Vor allem unter Unix bzw. Linux bieten z.B. Texteditoren wie der vi, der ed oder emacs die Möglichkeit in Datein mit solchen Audrücken zu suchen und Ersetzungen zu machen. Es gibt in diesem Metier auch Konsolen-Programme wie awk bzw gawk, die mit regulären Ausdrücken arbeiten.

Einige professionelle Texteditoren können ebenfalls mit Angabe von regulären Ausdrücken suchen und ersetzen. Zum Beispiel jEdit, Homesite, Notepad++, Crimson Editor und viele mehr.

In JavaScript gibt es auch Reguläre Ausdrücke.Dort sieht es so aus:

```
if ( text.match(/meine[fF]unktion/) ) {
    alert("Schreibfehler entdeckt");
}
```

5.Allgemeine Hinweise

Dass reguläre Ausdrücke cool und nützlich sind, haben wir ja bereits festgestellt. Jedoch sollte von der übermäßigen Benutzung abgesehen werden.

Ein regulärer Ausdruck wird erzeugt, kompiliert, gecached und auf eine Zeichenkette angewendet. Um einen regulären Audruck in C mit der pcre-library (die auch von PHP verwendet wird) anzuwenden muss man **ca 1 Seite C-Quellcode** schreiben. Dieser Aufwand versteckt sich in Scriptsprachen, sollte aber bei der Verwendung bedacht werden.

5.1.str_replace

Wenn **einfache Suchen/Ersetzen-Aufgaben** zu erfüllen sind, kann man ausweichen auf die Funktion str_replace (String-Replace).

Beispiel:

```
$string = str_replace(
                          'meineFunktion',
                          'MeineFunktion',
                          $string
                  );
// oder mehrere
$string = str_replace(
  array('meineFunktion','meinefunktion'),
  'MeineFunktion',
  $string
);
```

5.2.strpos

Um festzustellen, dass eine Zeichenfolge **in einem String vorkommt**, kann man alternativ auch die Funktion strpos (String-Position) verwenden. Kommt ein Ergebnis ungleich false heraus, dann liegt ein Treffer vor.

Beispiel:

```
if( strpos($text, 'spielen')!==false) {
     echo "Spielen kommt vor.";
}
```

5.3.substr

Muss beispielsweise am Anfang einer Zeichenkette „http://“ stehen, kann man das auch mittels substr (Sub-String) ermitteln:

Beispiel:

```
if( substr($text, 0, 7)=='http://') {
     echo "Fängt mit http:// an.";
}
```

6.Quick Reference

6.1.Steuerzeichen

\	Reserviertes Zeichen entwerten um danach suchen zu können.
\f	Form feed
\n	Zeilenumbruch, Newline
\r	Wagenrücklauf, Carriage return
\t	Tab
\e	Escape
\xxx	ASCII-Zeichen als Oktal-Zahl xxx
\xnn	ASCII-Zeichen als Hexadezimal-Zahl nn
\cX	Das Kontroll-Zeichen ^X. Z.B. \cI ist das selbe wie \t und \cJ ist das selbe wie \n. Werden zur Remote-Terminal-Steuerung verwendet.

6.2.Zeichen-Gruppen

[...]	Eines der Zeichen zwischen den Klammern muss vorkommen.
[^...]	Keines der Zeichen ziwschen den Klammern darf vorkommen.
.	Jeder Zeichen ausser \n. Das selbe wie [^\n]
\w	Jedes Wort-Zeichen. Das selbe wie [a-zA-Z0-9_] und [[:alnum:]_]
\W	Jeder Nicht-Wort-Zeichenr. Das selbe wie [^a-zA-Z0-9_] und [^[:alnum:]_]
\s	Jedes unsichtbare Zeichen (Whitespace, Steuerzeichen und Leerzeichen). Das selbe wie [\t\n\r\f\v] und [[:space:]]
\S	Kein unsichtbares Zeichen. Das selbe wie [^ \t\n\r\f\v] und [^[:space:]] Achtung: bedeutet nicht das selbe wie \w!
\d	Jedes Zahl-Zeichen. Das selbe wie [0-9] und [[:digit:]]
\D	Jedes Nicht-Zahl-Zeichen. Das selbe wie [^0-9] und [^[:digit:]]
[\b]	Ein Backspace (Zeichen-Rückschritt) (Spezialfall)
[[:class:]]	[:alnum:] [:alpha:] [:ascii:] [:blank:] [:cntrl:] [:digit:] [:graph:] [:lower:] [:print:] [:punct:] [:space:] [:upper:] [:xdigit:]

6.3.Ersetzungen

\\n	n-ten gruppierten Treffer (geklammert) in Ergebnis einsetzen. n ist eine Zahl von 1 bis 9, wobei 1 links startet. Bei Verschachtelung werden die umgebenden Klammer zuerst verarbeitet.

6.4.Wiederholungen

{n,m}	Treffe die davor definierte Gruppe mindestens n mal, aber maximal m mal.
{n,}	Treffe die davor definierte Gruppe mindestens n mal oder öfters.
{n}	Treffe die davor definierte Gruppe genau n mal
?	Treffe kein- oder einmal. Das selbe wie {0,1}
+	Treffe ein- oder mehrmals. Das selbe wie {1,}
*	Treffe die davor definierte Gruppe kein-mal oder mehrmals. Das selbe wie {0,}
+?	Non-greedy match.
*?	Non-greedy match. E.g. ^(.*?)\s*$ the grouped expression will not include trailing spaces.

6.5.Optionen

g	Macht eine globale Suche. Hört nicht nach dem ersten Treffer auf.
i	Ignoriert Gross-Kleinschreibung (case-insensitive)
m	Sucht über mehrere Zeilen. ^ and $ treffen dann pro Zeile zu.
s	Sucht als wär's alles in einer Zeile. ^ und $ ignorieren \n, aber . trifft \n.
x	Falls Kommentare und Whitespaces zur Übersichtlichkeit verwendet werden.

6.6.Erweiterter Regulärer Ausdruck

(?#...)	Kommentar, "..." wird ignoriert.
(?:...)	Trifft aber gibt "..." nicht zurück.
(?=...)	Trifft wenn der nachfolgende Ausdruck "..." treffen würde.
(?!...)	Trifft wenn der nachfolgende Ausdruck "..." nicht treffen würde.
(?imsx)	Ändern der Optionen mitten im Ausdruck.

6.7.Gruppierung

(...)	Mehrere Einzeilbausteine zusammenfassen. Auf die Gruppe kann *, +, ?, |, etc angewandt werden. Das Ergebnis solch einer Gruppe steht bei beim Ersetzen als Referenz zur Verfügung.
|	Alternierung / Entscheidung. Treffe entweder das linke oder das rechte.
\n	Trifft die selben Zeichen die getroffen wurden, als Gruppe n zum ersten mal getroffen wurde. Gruppen sind u.U. Verschachtelt und von aussen nach innen der Klammerung nummeriert von links nach rechts.

© 2009 Rüdiger Marwein