www.phparch.com

August 2018
Volume 17 - Issue 8

Masterful Code Management

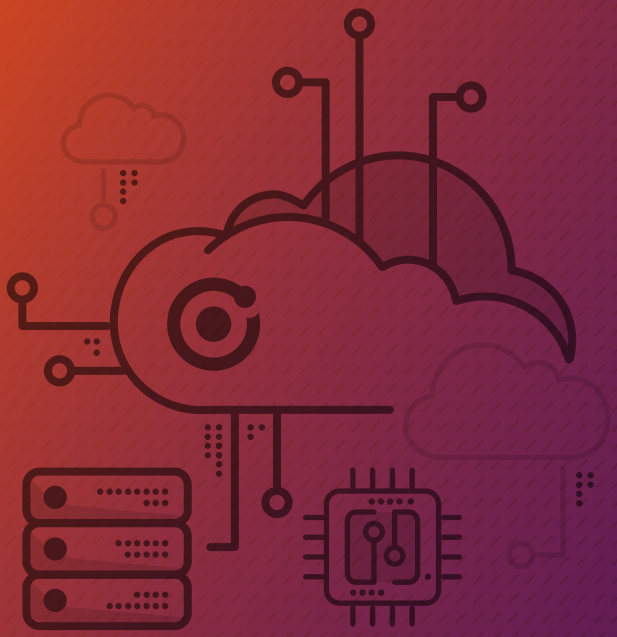
Debugging PHP With Xdebug

Succeeding as a Freelancer Developer

MySQL Generated Columns, Views, and Triggers

Pro Parsing Techniques With PHP, Part Three Using Regular Expressions

ALSO INSIDE



Start building with a focused, faster cloud.

Fully supported by Thermo Physicists there to help solve any of your problems.

thermo io

Focused

Elastic energy at your fingertips. Launch. Rebuild. Clone. Swap. Grow. Instantly.

Faster

Deploy in seconds on our high-availability, SSD-driven platform.

Care

Thermo.io Physicists are here to help you out without confiscating root.

DISCOUNT:

Get your $150 credit today using code

PHPDEVELOPER

WEB: *Thermo.io*

EMAIL: *Sales@Thermo.io*

PHONE: *833-3-THERMO*

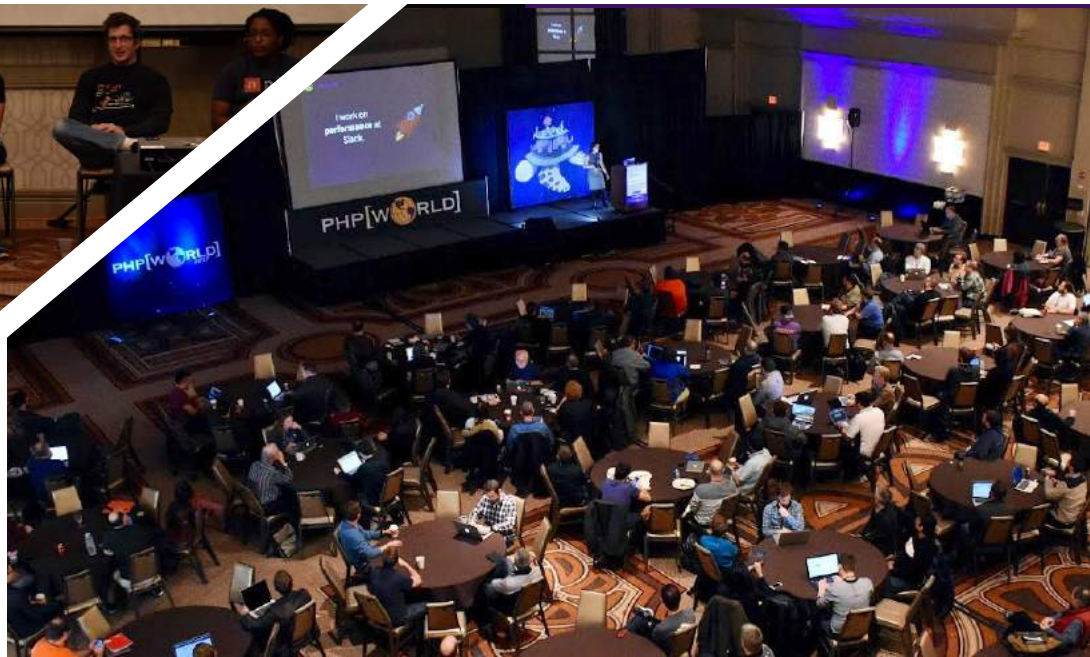
Save the dates to attend

PHP[WORLD] 2018

The 4th annual PHP conference for Washington D.C.

November 14–15, 2018
Washington, D.C.

world.phparch.com



CONTENTS

php[architect]

AUGUST 2018
Volume 17 - Issue 8

Features

Columns

Editor-in-Chief: Oscar Merida

Editor: Kara Ferguson

Managing Partners

Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by: musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Subscriptions

Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising

To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Pro Parsing Techniques With PHP, Part Three Using Regular Expressions

Michael Schrenk

This is the final installment of a set of three articles offering strategies for parsing text with PHP. The first article described the basics of parsing, followed by an article on developing fault tolerant parsing strategies. This article is dedicated to regular expressions.

Regular expressions, or sometimes simply called regex, represent a powerful set of tools which allow developers to split strings, perform character substitutions, and extract text based on matched patterns. The patterns, used in regular expressions, are an actual language that describe combinations of type castings and values that match the text you want to split, substitute, or extract. Regular expressions are an enormously powerful tool for the developer who understands them.

Scripts referenced though this series are available for download at http://www.schrenk.com/parsing.

PCRE and POSIX functions

There are two ways to use regular expressions in PHP, POSIX, or extended regular expressions, and Perl-Compatible Regular Expressions[1] (PCRE). The POSIX set of functions—which all started with `ereg_`—have been deprecated since PHP 5.3 and removed in PHP 7.0, so there's little need to mention them further. Instead, we'll focus our discussion to the PCRE version of regular expression functions. You can easily recognize the PCRE regular expression functions because they all start with the prefix "preg". Of these, there are five separate commands:

- `preg_replace()`,
- `preg_replace_callback()`,
- `preg_split()`,
- `preg_match()`, and
- `preg_match_all()`.

These functions allow developers to:

- substitute characters within a string when they match a predefined pattern,
- detect if a pattern of characters exists within a string, extract a string that matches a pattern,
- or split strings where a pattern is found.

Let's take a quick look at how these functions work.

For now, don't worry if you don't understand how patterns work. We'll explore that later.

preg_replace()

The first of the PHP regular expression functions we'll look at, `preg_replace()`, does a string substitution when characters in the input match a pattern, as shown below in Listing 1.

Once Listing 1 is executed, the word `new` will match and replace the pattern in `$subject`. The resulting value of $subject will be `This is a new string`. Keep in mind our pattern will also match the strings "tested", "testing", and even "pretest" as we'll see again later.

If you need more complicated matching logic, `preg_replace_callback` allows you to use another function to calculate and return the replacement string.

Listing 1

```
1. <?php
2. /*
3.  * Example: preg_replace() replaces one string with another,
4.  * when a pattern is matched.
5.  * USAGE: preg_replace($pattern, $replacement, input_string);
6.  * Where: $pattern is the pattern to match
7.  *        $replacement is the substitution string, and
8.  *        $subject is the source string.
9.  */
10. $pattern = '/test/';
11. $replacement = 'new';
12. $subject = 'This is a test string';
13. $parsed_string = preg_replace($pattern , $replacement, $subject);
```

1 *Perl-Compatible Regular Expressions: http://php.net/book.pcre*

preg_split()

This function splits a string at the point where the pattern is found, as shown below in Listing 2.

When Listing 2 is executed, `$before` will contain the string that was to the left of the pattern. So, it will hold the text, "`This is a`". The contents of `$after`, on the other hand, will contain that which is to the left of the pattern, or string.

preg_match()

The function `preg_match()` returns a Boolean value (0 or 1) depending on if the pattern is found in the subject string. In the example below, shown in Listing 3, `$bool` will be set to `true` because the pattern "`This`" is found in the input string.

The `preg_match()` function doesn't affect the original string. It only indicates if a pattern is found in the input string.

You can also pass an optional third parameter, an array, to `preg_match()`. If any part, or parts, of the pattern are found, the entire matching string will be returned in the first array element. Any other array elements will contain all individual, and subsequent, matches of the pattern.

preg_match_all()

The function `preg_match_all()` is essentially the same as `preg_match()`, but it always assumes you want to find all matches to the pattern. These functions mainly differ in that `preg_match()`, with two passed parameters, will stop after the first pattern match. So, if you only need to match the first match, `preg_match()` is the function to use.

An example of `preg_match_all()` is shown below in Listing 4.

In the above example, `var_dump($result_array)` will hold the following.

```
array (size=1)
  0 =>
    array (size=2)
      0 => string 'test' (length=4)
      1 => string 'test' (length=4)
```

Listing 2

```
1. <?php
2. /*
3.  * Example: preg_split() splits a string where a specific
4.  * pattern is matched.
5.  * USAGE: preg_split($pattern, $subject);
6.  * Where: $pattern is the pattern to match
7.  *        $subject is the source string.=
8.  */
9. $pattern = '/test/';
10. $subject = 'This is a test string';
11. list($before, $after) = preg_split($pattern, $subject);
```

Listing 3

```
1. <?php
2. # EXAMPLE: preg_match() returns a true/false dependent on
3. #  pattern being found in a string.
4. # USAGE: preg_match($pattern, $input_string)
5. # Where: $pattern is the pattern to match
6. #        $subject is the source string.
7. $pattern = '/test/';
8. $subject = 'This is a test string';
9. $bool = preg_match($pattern, $subject);
```

Listing 4

```
1. <?php
2. /**
3.  * Script 4
4.  * EXAMPLE: preg_match_all() returns every occurrence that
5.  *  matches the pattern
6.  * USAGE: preg_match_all($pattern, $subject, $result_array)
7.  * Where: $pattern is the pattern to match
8.  *        $subject is the source string.
9.  *        $result_array is returned matches.
10.  */
11. $pattern = '/test/';
12. $subject = 'This is a test of a test string';
13. preg_match_all($pattern, $subject, $result_array);
```

Differences From PHP Built-In Functions

You may notice these regular expression functions are very similar to several PHP built-in functions. For example, `preg_replace()` is very similar to `str_replace()`. The function `preg_split()` is much like `substr()` and `explode()`, while `preg_match()` is nearly the same as `strstr()`. Furthermore, `preg_match_all()` is very similar to the `parse_array()` command, which is not a PHP built-in function, but was described in the first article of this series. Again, functions used in the first article are available for download[2].

While these functions are similar, they differ in one very important way. The regular expression functions are not limited to matching a simple string pattern like "`test`". The patterns used in regular expressions are capable of defining patterns that might:

- match anything that's a number,
- match any alphanumeric characters,
- Match ranges of characters,
- match any characters of a specific length,
- match words, of a specific length, that start, end or contain a specific pattern.

More so, regular expression patterns can be combined to match just about any conceivable pattern. It's the ability to match patterns of various types which makes regular expressions so rich and powerful.

2 *download: http://www.schrenk.com/parsers*

What Are Patterns?

Regular expression patterns are a language specifically designed for pattern definition. In the previous examples, our pattern was a specific set of characters, like `/test/`. When this is the case, our regular expression will match all occurrences of the string `"test"`, even if it exists in larger sets of characters like `"retest"` or `"testing"`.

There are more ways to create regular expression patterns than what will fit in this article. So, we'll use this space to introduce you to just enough techniques so you can thoughtfully explore more complete documentation. (An excellent online tutorial on regular expression patterns can be found at on php.net under Pattern Syntax[3] We will build toward a test case, where we build a pattern which helps us extract all phone numbers from a document.

Perhaps the first thing you'll notice about patterns is that they are typically encased between backslashes.

```
/pattern/
```

These backslashes are not to be confused with slashes, which define special characters. For example, the pattern `/\d/` will match any single digit. The pattern `/\d\d\d/` will match any three consecutive digits. Notice I didn't say it will match any three-digit number, because it will also match any three consecutive digits in a larger number as well. If we wanted to match any sequence of three random non-digit characters, we could use the pattern `/\D\D\D/`, or simply `/\D{3}/`.

These patterns can get quite complex. The combination of slashes and backslashes, also known as the "leaning toothpick syndrome," can be hard to look at. Rather than focusing on all the options, let's learn a few patterns by using them in an example application.

Using Regular Expressions to Extract Phone Numbers

Regular expressions are perhaps most useful when you need to extract arbitrary information from a block of text. For example, patterns are useful when parsing all the phone numbers from a document. I may be taking some liberties by calling phone numbers arbitrary because obviously, phone numbers do follow one of several specific patterns. But they are also arbitrary in that the phone numbers are not associated with a specific person, time, or another context.

The first step in developing a pattern that will match phone numbers is to decide what pattern, or patterns, must be matched. The definitions we'll look at only relate to North American phone numbers. Phone numbers from other parts of the globe may be formatted differently. Our phone numbers tend to follow the specific pattern below:

- `aaa d ppp d nnnn`
- `aaa` is the three-digit area code
- `d` is some type of delineator
- `ppp` is a three digit prefix, and
- `nnnn` is a four digit line number.

Now that we have developed a basic template for what we plan to match, let's create a test case. For example, the phone numbers below in Listing 5 are a

3 *Pattern Syntax:* http://php.net/reference.pcre.pattern.syntax

good start.

Let's examine the pattern in Listing 5. You're already familiar with the slash characters at the start end of the pattern. You should also recognize the `\d{3}` and `\d{4}` patterns that match three and four digit numbers respectively. Additionally, the `\D` pattern matches any non-digit character.

When you put this pattern together, you're essentially saying, "Extract anything that matches three digits, followed by any alpha character, followed by three more digits, another alpha character, and four more digits."

As it is, our initial pattern matched only four of the numbers, `"111 222 3333"`, `"100 222-3333"`, `"111.222.3333"`, and `"100 222-3333"`. What's missing are the phone numbers that contain parentheses and the one that contains unformatted numbers. To match those, we will create two new patterns and then combine them with the initial pattern we developed.

This final script correctly extracts all the phone numbers in our test string by adding two new patterns, and then combining all three with a Boolean OR operator. You'll notice in the second pattern, the parentheses are added, but they are preceded by a backslash to indicate a literal character, or that the open and close parentheses are the only acceptable matches. This is because parenthesis can be used to group patterns within a regular expression.

The third pattern simply says "match on any sequence of exactly 10 digits". Combining separate patterns allows patterns to be debugged independently. Also, combining debugged patterns also makes the whole pattern easier to read. For example, if one pattern were defined, it would be much harder to debug and would look like this.

```
/\d{3}\D\d{3}\D\d{4}|\(\d{3}\)\D\d{3}\D\d{4}|\d{10}/
```

Final Thoughts

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*
>
> *–Jamie Zawinski,*

This is a very quick tour of regular expressions. Hopefully, it will provide a foundation where you can explore more on your own. Regular expressions are an incredibly powerful set of commands for pattern matching, string splitting, and character substitution. So, it would appear that they are particularly ideal for parsing tasks. Regular expressions are so powerful that—for many developers I've talked to—they are the primary text parsing tool. Regular expressions, however, are a double-edged sword. While it's true they're powerful, regular expressions are also complex and with many subtle options. So while you might get a lot of parsing from a single regular expression, you'll also end up with scripts that are difficult to read. And, anything that is difficult to read is also difficult to debug, especially compared to the same parse completed with a handful of PHP commands.

When Is the Best Time to Apply Regular Expressions?

Regular expressions are the perfect choice for applications like the previous phone number extraction example. You should be cautioned, however, to limit the use of regular expressions to cases where the basic functions in the first article of this series are insufficient for the task. I first made this comment twelve years ago in the first edition of my book, *Webbots, Spiders, and Screen Scrapers, First Edition* (2004, No Starch Press, San Francisco). At the time, I didn't realize how controversial it would be to question if regular expressions should be recommended for every parsing task. But after

Listing 5

```
1. <?php
2. // Create a test string containing phone numbers
3. $string = "Example #1, 111 222 3333,   Example #2,  100 222-3333,
4.            Example #3, 111.222.3333,   Example #4,  <td>100 222-3333</td>,
5.            Example #5, (111) 222 3333, Example #6,  (111) 222-3333,
6.            Example #7, (111) 222.3333, Example #8,  1112223333";
7.
8. // Define a pattern
9. $pattern = "/\d{3}\D\d{3}\D\d{4}/";
10.
11. // Run the regular expression
12. preg_match_all($pattern, $string, $matching_numbers);
```

Listing 6

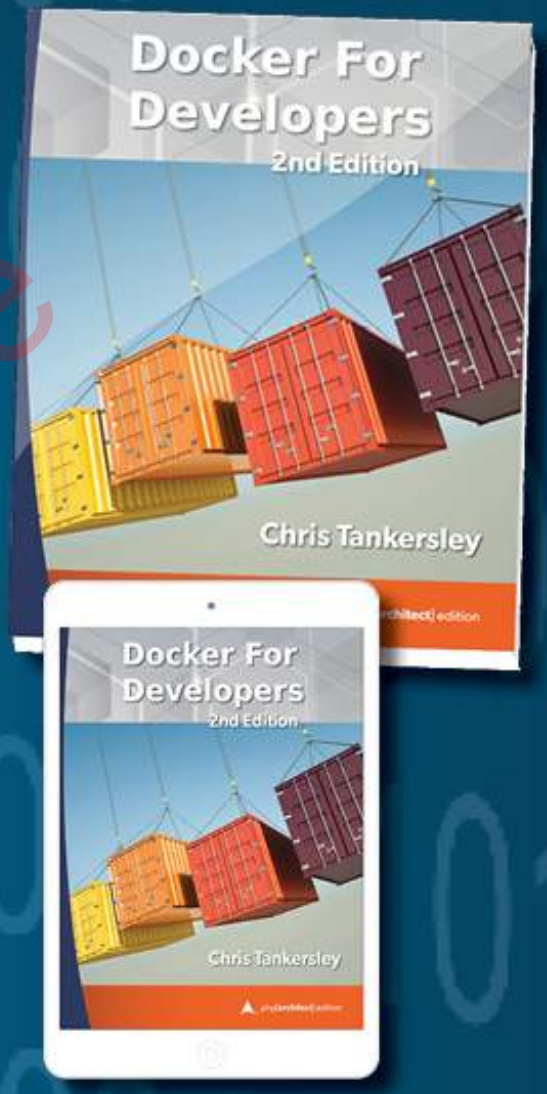
```
1. <?php
2. // Test string containing phone numbers
3. $string = "Example #1, 111 222 3333,   Example #2,  100 222-3333,
4.            Example #3, 111.222.3333,   Example #4,  <td>100 222-3333</td>,
5.            Example #5, (111) 222 3333, Example #6,  (111) 222-3333,
6.            Example #7, (111) 222.3333, Example #8,  1112223333";
7.
8. // Define patterns
9. $pattern_1 = "\d{3}\D\d{3}\D\d{4}";
10. $pattern_2 = "\(\d{3}\)\D\d{3}\D\d{4}";
11. $pattern_3 = "\d{10}";
12. $pattern_combined = "/(" . $pattern_1 . "|" . $pattern_2
13.            . "|" . $pattern_3 . ")/";
14.
15. // Run the regular expression
16. preg_match_all($pattern_combined, $string, $matching_numbers);
```

Read a Sample Online

Docker For Developers is designed for developerswho are looking at Docker as a replacement for development environments like virtualization, or devops people who want to see how to take an existing application and integrate Docker into that workflow.

This book covers not only how to work with Docker, but how to make Docker work with your application. This revised and expanded edition includes:

- creating custom images,
- working with Docker Compose and Docker Machine,
- managing logs,
- 12-factor applications.



Order Your Copy

http://phpa.me/docker-devs

receiving a host of unanticipated emails from angry regular expression aficionados, I backed off a bit on my prejudices. A few years later, I wrote an additional chapter for the second edition of the book, solely on the effective use of Regular expressions. And while I've updated my biases accordingly, I still feel strongly on the limited used of regular expressions.

> *PHP has built-in parsers for XML and HTML via DomDocument and SimpleXML which can be more fault tolerant especially for complicated XML or malformed HTML. If you need to parse JSON strings,* `json_decode` *should be your tool of choice instead of regular expressions. Regular expressions shine when your text is less structured.*

Taming Complexity

The power of regular expressions comes at the cost of simplicity. In most software development, the simplest approach is usually the easiest to develop, read, debug, and maintain. And while a very complex regular expression pattern could take a day to develop and two more days to document, it doesn't mean regular expressions need to be hard to use. If you've looked at the code from the first article in this series, you'll notice the function `parse_array()` uses regular expressions. That function, however, hides the inner workings of regular expressions in a wrapper function, or a function that simply repackages an existing function to make it more readable. This wrapper function accomplishes two things. First, it makes your code much easier to read. For example, if you wanted to parse all the image tags from a webpage (contained in the variable `$webpage`), you could use either of the following scripts.

Both methods in Listing 7, return the same values in the array `$images`. The second technique, however, has a few advantages. First, the second method is much easier to understand, especially if you are not familiar with the language regular expressions. And as mentioned earlier, easier to understand usually translates easier to debug, document, and maintain. Not only does the wrapper function only use a single line of code in your script, but it also spares you from needing to remember what `siU` stands for.

Listing 7

```
1. <?php
2. /**
3.  * Get image references from a web page using regular expressions.
4.  * Where: $www = a complete webpage
5.  */
6.
7. // With regular expressions
8. preg_match_all('/(<img(.*)>)/siU', $www, $matching_data);
9. $images = $matching_data[0];
10.
11. // With the parse_array() function
12. $images = parse_array($www, "<img", ">");
```

Regular expressions, while powerful, can also be difficult for people, who are less familiar with regular expressions, to read. It's worth repeating—difficult to read means difficult to develop. Sometimes that fact doesn't matter, because the parse is complex, and regular expressions are the only method for extracting information. The other reason for using a wrapper function instead of the direct regular expression is the wrapper helps focus the developer on the task at hand. Instead of trying to remember how the regular expression is written, a more human-readable, debugged, wrapper function can be used.

Where Regular Expressions Shine

The problem with regular expressions is that, with just a pattern match, they operate without context in which the data exists. For example, in our phone number parsing example, we didn't need to know if those phone numbers belonged to anyone. If they had, the context would be lost. But in cases where simple (and sometimes not so simple) patterns need to be matched, regular expressions are irreplaceable. But, as mentioned in the first article in this series, I highly recommend exploring simpler parsing strategies first.

Thanks for staying with me for this series of articles on parsing. If you have a comment, please drop me a line at mike@schrenk.com.

Michael Schrenk has developed software that collects and processes massive amounts of data for some of the biggest news agencies in Europe and leads a competitive intelligence consultancy in Las Vegas. He consults on information security and Big Data everywhere from Moscow to Silicon Valley, and most places in between. Mike is the author of Webbots, Spiders, and Screen Scrapers (2012, No Starch Press). Mike is also an eight-time speaker at the notorious DEF CON hacking conference. @mgschrenk

Related Reading

- *RegEx is Your Friend* by Liam Wiltshire. July 2016 issue. https://phparch.com/magazine/2016-2/july/
- *Pro Parsing Techniques With PHP, Part One: Simplifying Your Parsing Strategy* by Michael Schrenk. June 2018 issue https://www.phparch.com/magazine/2018-2/june/
- *Pro Parsing Techniques with PHP, Part Two: Fault Tolerance* by Michael Schrenk. July 2018 issue https://www.phparch.com/magazine/2018-2/july/

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[**architect**]

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe