Functional Programming in PHP

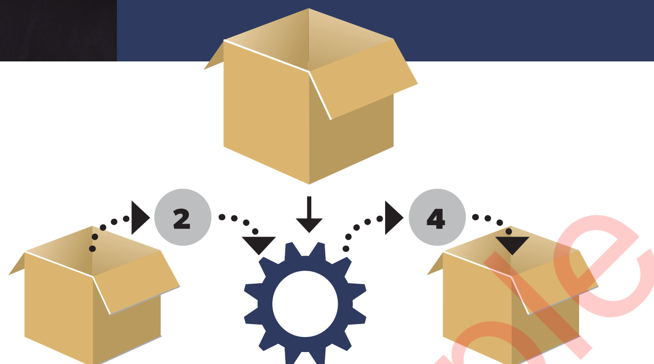
Second Edition

by Simon Holywell

Sample

a php[**architect**] guide

Simon Holywell is a Senior Software Engineer at Aurion in Brisbane, Australia (http://aurion.com) and is passionate about web application development and motorcycles. His first public project was written with PHP 3, and since then, he has worked with every version of PHP and dabbled in Python, Scala, C, JavaScript, and more. He is also the author of of SQLStyle.guide and the ssdeep extensions for PHP's PECL, Facebook's HipHop Virtual Machine (HHVM), and MySQL.



Many languages have embraced Functional Programming paradigms to augment the tools available for programmers to solve problems. It facilitates writing code that is easier to understand, easier to test, and able to take advantage of parallelization making it a good fit for building modern, scalable solutions.

Sample

PHP introduced anonymous function and closures in 5.3, providing a more succinct way to tackle common problems. More recent releases have added generators and variadics which can help write more concise, functional code. However, making the mental leap from programming in the more common imperative style requires understanding how and when to best use lambdas, closures, recursion, and more. It also requires learning to think of data in terms of collections that can be mapped, reduced, flattened, and filtered.

Functional Programming in PHP will show you how to leverage these new language features by understanding functional programming principles. With over twice as much content as its predecessor, this second edition expands upon its predecessor with updated code examples and coverage of advances in PHP 7 and Hack. Plenty of examples are provided in each chapter to illustrate each concept as it's introduced and to show how to implement it with PHP. You'll learn how to use map/reduce, currying, composition, and more. You'll see what external libraries are available and new language features are proposed to extend PHP's functional programming capabilities..

a php[architect] guide
www.phparch.com

Functional Programming in PHP

Second Edition

by

Simon Holywell

Sample

a php[architect] guide

Functional Programming in PHP—a php[architect] Guide

Contents Copyright ©2016 Simon Holywell—All Rights Reserved

Book and cover layout, design and text Copyright ©2016 musketeers.me, LLC. and its predecessors – All Rights Reserved

Second Edition: October 2016
ISBN - print: **978-1-940111-46-9**
ISBN - PDF: **978-1-940111-47-6**
ISBN - epub: **978-1-940111-48-3**
ISBN - mobi: **978-1-940111-49-0**
ISBN - safari: **978-1-940111-50-6**

Produced & Printed in the United States

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by means without the prior written permission of the publisher, except in the case of brief quotations embedded in critical reviews or articles.

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided "as-is" and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavored to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

musketeers.me, the musketeers.me logo, php[architect], the php[architect] logo, php[architect] Guide are trademarks or registered trademarks of musketeers.me, LLC, its assigns, partners, predecessors and successors. All other trademarks are the property of the respective owners.

Sample

Written by
Simon Holywell

Published by
musketeers.me, LLC.
201 Adams Ave.
Alexandria, VA 22301 USA

240-348-5PHP (240-348-5747)
info@phparch.com
www.phparch.com

Editor-in-Chief
Oscar Merida

Technical Reviewers
Koen van Urk and Oscar Merida

Copy Editor
Kara Ferguson

Layout and Design
Kevin Bruce

Table of Contents

	About the Author	**VII**
	Acknowledgements	**XI**
Chapter 1.	**Introduction**	**1**
	Prerequisites	1
	Requirements	2
	For Users of Older PHP Versions	2
	Installing	3
Chapter 2.	**What is Functional Programming?**	**7**
	Let's See Some Code	9
	History	9
	Other Functional Implementations	15
	Commercial Uses	15
	What is Functional Programming Best for?	16
	The Benefits of Functional Programming	16
	Functional Basics	17

Sample

Chapter 3. Language Features 19
Types 20
Functions 23
Namespacing 39
Recursion 40
Map, Reduce, and Filter 42
Memoization 47
Generators 47

Chapter 4. Helpful Libraries 49
Library Installation 50
The iter Library 52

Chapter 5. HHVM's Hack 55
Types 56
Lambda Expressions 65
Special (Magical) Attributes 66
Conclusion 67

Chapter 6. Patterns 69
Head and Tail 70
Flattening lists 70
Handling Your NULLS 72
Composition 75
Partial Functions 76
Pipelines 78
Pattern Matching 80
Functors 83
Applicatives 88
Monads 93

Sample

Chapter 7. Implementing the Theory 103

IP Address Restriction 103
Functional Primitives 105
A Domain Specific Language in PHP 108

Chapter 8. Event Driven Programming 117

ReactPHP Installation 118
Getting Started 118
Add Some Logging 119
Introduce a Monad 119
Callback Wrangling 122
Wrap Up the Show 129

Chapter 9. Hazards of Functional Programming in PHP 131

Chapter 10. Advances in PHP 133

PHP 5.4 133
PHP 5.5 134
PHP 5.6 135
PHP 7 137
Further into the Future 140

Chapter 11. Conclusion 143

Appendix A. Additional Notes 145

Understanding Type Signatures 145
Using the UTF-8 Ellipsis 149

Appendix B. Resources 151

PHP REPLs 151

Sample

Libraries 152
Other Functional Implementations 152
Online Courses (MOOC) 152

Glossary 153

Index 159

Sample

About the Author

Simon Holywell is a Senior Software Engineer at Aurion in Brisbane, Australia (*http://aurion.com*) and is passionate about web application development and motorcycles. His first public project was written with PHP 3, and since then, he has worked with every version of PHP and dabbled in Python, Scala, C, JavaScript, and more. He is also the author of SQLStyle.guide, and the ssdeep extensions for PHP's PECL, Facebook's HipHop Virtual Machine (HHVM), and MySQL.

Blog: *https://www.simonholywell.com*
Twitter: *@Treffynnon*—*http://twitter.com/Treffynnon*

Sample

Sample

Chapter 6

Patterns

"Monads are return types that guide you through the happy path."
–Erik Meijer (Computer Scientist, *@headinthebox*)

Sample

In functional programming there are repeating patterns just as in any other programming style. This is in fact a major source of interest in functional programming from unacquainted coders, as it brings with it concise code.

Software patterns are generally agreed best practices for completing similar tasks in a universally identifiable and understood way. Much like object oriented programming, functional code also has a number of patterns you will regularly see. These patterns assist in making more composable functions allowing for greater reuse across various problem domains. Not only this, but they can help to make your API more consistent for those implementing any functional library you might have written. If the pattern is followed then implementers can accurately anticipate the result when they call the pattern from their code.

Patterns can be reproduced across a large number of types easily and quickly in most functional languages, but in PHP we need to lay the ground work ourselves first. Looking to the world of mathematics there are far more patterns than we will cover here. Some patterns are borrowed from the ideas implemented in the Haskell programming language, which can be a little difficult to reproduce given PHPs weak type system. It is, however, possible and they help to present excellent ways of producing reusable code.

It does not mean we will avoid implementing difficult patterns though with applicatives and monads both discussed. Some might suggest only a Haskell programmer should be interested in these formalisms, but they would be short sighted. It is easier to learn a pattern in a language you already know well than a completely foreign one. And, the patterns are actually useful in PHP code too, as we will explore further into the book.

Reusable patterns are great and what we all strive for as programmers. Good, functional code takes this up a notch and tries to abstract all operations into reusable and immutable code. To this end we will now explore some common and some complex patterns in functional PHP code.

Should you be unwilling to push the boundaries of PHP, turn back now and wallow in your safe billabong (oxbow lake, resaca, bayou). The next section has a raft and it is headed for the white water!

Head and Tail

We will start with a very simple pattern first though to ease our way into it. When working with lists and recursion it can be very helpful to be able to easily obtain the first element of an array (the head) and what is known as the tail of the array. Head, given an array, will return the first value from that array.

```
function head(array $arr) {
   return reset($arr);
}
head([1,2,3,4,5]); // 1
```

At the other end of the equation we have tail that will return a list with all but the first value in the array contained in it.

```
function tail(array $arr) {
   return array_slice($arr, 1);
}
tail([1,2,3,4,5]); // [2,3,4,5]
```

These two functions can be used together to work through a list using recursion.

```
function print_items(array $arr) {
   echo head($arr) . '-';
   if(tail($arr)) print_items(tail($arr));
}
print_items([1,2,3,4,5]); // 1-2-3-4-5-
```

Sample

Flattening lists

Lists of lists can be very helpful when dealing with complex datasets or when transforming an array via `array_map()` where it would return an array from the applied function. In some instances though, you have a list of lists that really should just be one list with all values at the top level. The problem could look something like:

```
$arr = [1, 2, 3, 4, 5];
$divisor = 10.5;
$arr2 = array_map(function($x) use ($divisor) {
   return [$x, $x / $divisor, $x % $divisor];
}, $arr);
```

Continued Next Page

```
// [
//    [1, 0.095238095238095233, 1],
//    [2, 0.19047619047619047, 2],
//    [3, 0.2857142857142857, 3],
//    [4, 0.38095238095238093, 4],
//    [5, 0.47619047619047616, 5]
// ]
```

Now you need to `array_sum()` all the values, but you have a multi-dimensional array. You want to flatten your list.

```
$arr3 = flatten($arr2);
// [
//    1, 0.095238095238095233, 1, 2, 0.19047619047619047, 2,
//    3, 0.2857142857142857, 3, 4, 0.38095238095238093, 4,
//    5, 0.47619047619047616, 5
// ]
```

After flattening the array all values are at the top level of the array and it is no longer multi-dimensional so we are now able to perform that all important `array_sum()` call.

```
array_sum($arr3); // 31.428571428571
```

PHP does not come with a `flatten()` function by default—as you might have suspected—and we are going to have to write one ourselves as in Listing 6.1. To make it more flexible we will add a maximum depth argument so an implementer can decide how many levels of their array they want to be flattened—starting from the top most dimension. Additionally it is often very useful for associative array keys to be maintained after the transformation so index access is not affected.

Listing 6.1

```
01. function flatten(array $array, $max_depth = null, $curr_depth = 1) {
02.    $out = [];
03.    foreach($array as $key => $val) {
04.       if(is_array($val)) {
05.          if(is_null($max_depth) || $curr_depth < $max_depth) {
06.             $val = flatten($val, $max_depth, $curr_depth + 1);
07.          }
08.          $out = array_merge($out, $val);
09.       } elseif(is_int($key)) {
10.          $out[] = $val;
11.       } else {
12.          $out[$key] = $val;
13.       }
14.    }
15.    return $out;
16. }
```

This definition of `flatten()` will maintain associative keys and reset integer keys. It will flatten all dimensions of an array to one unless `$max_depth` specifies otherwise. It is common to only flatten an array by one level so a helpful function to have on hand is `flatten_one()`, which can be written in terms of `flatten()`.

```
function flatten_one(array $array) {
   return flatten($array, 1);
}
```

Sample

This is more obvious and easier to read when reviewing code than simply using `flatten($arr, 1)` all over the place in code (it is easier to search/`grep` for too).

Now that we have a working `flatten_one()` implementation, we can perform an `array_map()` with keys in a much easier way than before by making use of a closure to handle the requirement for keys.

```
function map_with_keys(array $array, callable $func) {
    $ks = array_keys($array);
    $fx = function($key) use ($array, $func) {
        return [$key => $func($key, $array[$key])];
    };
    return flatten_one(array_map($fx, $ks));
}
```

Instead of trying to work through the values of the array we can iterate over the keys above and then obtain the value later using the key inside `$fx`. To ensure that keys are maintained during the operation `$fx` returns an associative array that is later flattened using `flatten_one()`.

```
map_with_keys(
   ['a' => 1, 'b' => 2],
   function($k, $v) {
      return $v . $k;
   }
); // ['a' => '1a', 'b' => '2b']
```

With the keys being passed to our callback function it is now possible to incorporate the key into reduce operations and make use of it.

```
reduce_with_keys(
   ['a' => 1, 'b' => 2],
   function($acc, $k, $v) {
      return $acc . $k . $v;
   },
   ''
); // a1b2
```

As you can see in the highly contrived example above, the array is reduced through string concatenation. The resultant string contains both the keys and the values of each element within the array.

Handling Your NULLS

There are a number of functions in PHP—and I am sure many more in the legacy userland code you work on—returning `NULL` when no record can be found, for example. When you then call the function, you cannot be sure if it will return a record as you expect or a null value, which violates the principle a function should always return the same type so it can be handled in the same way.

We have previously implemented PHP's `array_reduce()` function, and it serves again as a great example. When it is fed an empty array, it will return `NULL`. Imagine the result of our reduce operation were to be passed into a function expecting to receive an integer, then it would trigger an error from PHP's parser. There are a few ways to handle these null values and protect your code from unforeseen errors.

In the case of `array_reduce()`, you can simply set an initial value as the third parameter to the function. This initial value will be returned if the input array is empty. Otherwise, it will be used as the base value to add each reduce operation to.

If you do not have control over the code you are calling or you are implementing a function that doesn't allow for a default return value, then you can make use of the following two techniques.

Sample

Index

A

algorithms, 2, 17, 23, 33, 35
annotations, 60, 66
applicatives, 70, 88–94, 148
  context, 91–92
  interface, 89–90
  type class, 89–90
arguments
  list, 28
  name, 147
  supplied, 23, 75
array
  associative, 72
  class notation, 133
  dereference, 135
  filter, 104
  notation, 81
  sorted, 105
Async, 122–24
asynchronous operations, 124, 128
autoloading, 40, 132

B

Babbage, Charles, 1
Barclays Bank, 16
Bletchley Park, 12–13
Boole, George, 10
boolean algebra, 10

C

callbacks, 32, 35, 44, 46, 49, 122, 126, 133
Callback Wrangling, 122–23, 125, 127
cast operator, 21
class
  instance, 36–38
  name, 25–26, 38
closures, 20, 24, 26, 29–34, 36–38, 44–45, 65, 72, 117, 119, 133–34, 140, 144, 152–53, 157
  arguments, 34
code
  asynchronous, 117, 126
  imperative, 15, 143
  object oriented, 15, 39
composer, 2–4, 52, 75, 105, 118, 122, 124, 156
  autoload, 4, 52, 105, 118
  automation, 3
  dump-autoload, 4
  init, 3, 118
  install, 4, 51, 118
  update, 51
  vendor, 3, 39
composition, 36, 75, 90, 100
constructor, 63, 95, 111
currying, 12, 36, 76–78
  functionality, 77
  process, 77

D

dependencies, 3–4, 50–53
domain specific languages (DSL), 15–16, 108, 115, 154
DSL. See domain specific languages

E

Elixir, 79
Enigma code, 12
Enum, 59
Erlang, 14, 16
event, 117–18, 120, 122, 124, 126, 128, 130
  loop, 118–19, 127–28
Everest, George, 10
extensions, 2–5, 26, 28, 35, 41, 49–50, 56, 117, 154, 158
  functional-php, 50
  php-immutable, 132

F

Facebook, 3, 5, 55–56
factory, 111, 118
Flat Map, 45–46
flatten, 46, 71–72
Flowers, Tommy, 12
fmap, 84–90

Sample

Frege's Basic Law, 10
function
  anonymous, 28–29, 33, 41, 65, 82, 93, 97, 153, 155
  arguments, 26, 28, 56, 60, 81, 136, 138, 147, 154
  basic monad, 120
  body, 24–25
  callback closure, 107
  composable, 69, 100
  compose, 50, 100–101
  higher order, 14
  list/iterable access, 52
  named, 23, 25, 28, 121
  namespacing, 136
  objects, 36, 38, 42, 126, 154
  parameters, 27, 59–60
  pipeline, 78–80
  recursive, 11–12, 40
  return values, 135, 138
  variadic, 135, 149
  wrapped, 88–89, 93
functional languages, 7, 14–16, 29, 69, 93, 131, 147
FunctionalPhp, 140
Functionals, 47, 78, 105, 144
functors, 36, 38, 83, 85–89, 91, 93–94, 96, 134, 148
  law, 87

G

Geheimschreiber, 12
generator, 47–48, 52–53, 134, 154–55
  expressions, 140
  syntax, 47
generics, 59, 62–64
Gödel, Kurt, 11–12

H

Hack, 6, 55–62, 64–67, 154
  language, 3, 55–56
hash, 33, 47
Haskell, 8, 14–16, 80–81, 84, 94, 98, 143, 148, 152, 154
  programming language, 69, 78
  types, 147
HHVM's Hack, 55–56, 58, 60, 62, 64, 66, 68

I

iterators, 48, 52

J

JavaScript, 29, 49, 117
Java Virtual Machine (JVM), 15, 56, 155, 157
Jones, Simon Peyton, 8, 103
JVM. See Java Virtual Machine
Jython, 56

K

key/value list, 20, 58

L

lambda, 28–29, 41, 117, 121, 144, 153, 155
  calculus, 11, 29
  expressions, 11–12, 65–66
  functions, 18, 20, 28–34, 41, 65, 133
lazy loading, 29, 32, 75
libraries
  functional, 50, 69
  php-option, 74, 98
LISP, 13–14, 16
  programming language, 133
logging, 100, 119
Lovelace, Ada, 1

M

map, 14, 32, 35, 42–46, 52–53, 58, 62–63, 70, 72, 76, 79–80, 94–97, 99, 102, 119–20
  ordered, 20, 58
memoize, 47, 66
Monadic Laws, 96
monads, 69–70, 93–99, 101–2, 119–22, 148, 155
  container, 120
  interface, 94
  list, 120
  state, 102
  structure, 94
  writer, 99, 102

Sample

N

namespaces, 25
NET framework, 154
Newman, Max, 12

O

object-oriented programming, 16, 29
OCaml, 56, 152, 155
operator, splat, 27–28, 136

P

partial function application, 36, 50, 76–77, 82, 156–57
pattern matching, 15, 80–82
PDO, 98–99
PECL, 2, 5, 50, 156
  extension, 4–5, 49, 117
PHPDaemon, 129
PHPDoc, 145–46, 156
PHP SuperClosure, 29, 152
pipelines, 78–80, 156
primitives, functional, 49–50, 105, 107–8, 154
promises, 124, 126–28
  deferred, 127–28
PsySH, 2, 151
Python, 14, 41, 56

R

React/Async project, 124
React/Partial, 2, 50–51, 76, 149, 152, 157
ReactPHP, 117, 119, 129, 143
recursion, 11, 14, 19–20, 40–41, 44, 70, 157
  direct, 41
  indirect, 41
  infinite, 155
  mutual, 41
recursive form, common, 40
REPL, 2, 156–57
resolver, 125, 127–28
RFC, 66, 106, 135, 138, 140, 144, 157
Russell, Bertrand, 10

S

Scala, 15–16, 49, 80–81, 143, 147–48, 152, 157
  Option values, 74
  programming language, 15, 152
scope, 24, 30, 36–38, 65, 128
  parent, 24–25, 157
state
  avoiding, 117
  global mutable, 7, 41
  maintaining, 9
subroutines, 153–55
superglobals, 24

T

transparency, referential, 8, 17, 157
Turing, 12–13, 16
  Machines, 11–12
  Test, 13
type hints, 23, 25–26, 59, 86, 136–38
  callable, 133
  scalar, 137
  strict, 23
types
  complex, 88
  enumeration, 59
  expected, 147
  Hack, 56
  immutable, 59
  internal, 60
  new, 56, 61
  scalar, 20, 26, 56, 147
  strict, 137
  weak, 69

U

Ubuntu, 3, 5–6, 158
Underscore.php, 50–51, 132, 144
US Secure Hash Algorithm, 33
UTF-8, 113, 121, 158
  Ellipsis, 76, 149, 158

Sample

V

values
carried, 79
coerce, 147
primitive, 43
scalar, 111
variables, global, 25, 30
variadics, 26–28, 135–36
Vector, 58, 62

W

Windows, 6, 149, 158

Z

Zend, 157
Engine, 56
runtime, 139
Zephir, 56

Sample

php[architect] Books

The php[architect] series of books cover topics relevant to modern PHP programming. We offer our books in both print and digital formats. Print copy price includes free shipping to the US. Books sold digitally are available to you DRM-free in PDF, ePub, or Mobi formats for viewing on any device that supports these.

To view the complete selection of books and order a copy of your own, please visit: *http://phparch.com/books/*.

- **Web Security 2016**
  Edited by Oscar Merida
  ISBN: 978-1940111414

- **Docker for Developers**
  By Chris Tankersley
  ISBN: 978-1940111360 (Print edition)

- **Building Exceptional Sites with WordPress & Thesis**
  By Peter MacIntyre
  ISBN: 978-1940111315

- **Integrating Web Services with OAuth and PHP**
  By Matthew Frost
  ISBN: 978-1940111261

- **Zend Framework 1 to 2 Migration Guide**
  By Bart McLeod
  ISBN: 978-1940111216

- **XML Parsing with PHP**
  By John M. Stokes
  ISBN: 978-1940111162

- **Zend PHP 5 Certification Study Guide, Third Edition**
  By Davey Shafik with Ben Ramsey
  ISBN: 978-1940111100

- **Mastering the SPL Library**
  By Joshua Thijssen
  ISBN: 978-1940111001

Sample