www.phparch.com

July 2017
VOLUME 16 - Issue 7

php[architect]

# Safe at Speed

The Train Wreck: When Safety Is Discretionary

Smart, Scalable Content Distribution

Zero to Cloud in One Hour With the Google Cloud

Modern JavaScript: Moving Beyond jQuery

## ALSO INSIDE

Free Sample Article

**Education Station:**
Simple, Compact Time Range Creation with Period

**Community Corner:**
Lend More Than Your Voice

**Artisanal:**
Forms And Request Processing

**The Dev Lead Trenches:**
So Now You're a Team Lead

**finally{}:**
On Mental Health—My Personal Story

# PHP[WORLD] 2017

Early Bird Pricing until August 4th!

**This is a conference like no other. Designed to bring together all the communities linked by the PHP programming language.**

**November 15-16, 2017**
**Washington, D.C.**

# world.phparch.com

Sponsored by:

Magento®

MySQL™

php[architect]

# CONTENTS

JULY 2017
Volume 16 - Issue 7

# Safe at Speed

## Features



## Columns





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



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

**Print ISSN** 1709-7169
**Digital ISSN** 2375-3544

# Simple, Compact Time Range Creation with Period

*Matthew Setter*

**For the longest time, I've enjoyed using PHP's DateTime library[1]. I've always found it to be relatively straightforward in creating `DateTime` objects for use with various applications I've written.**

**However, one thing that isn't very simple, nor intuitive, is the ability to create time ranges—especially ones requiring some degree of sophistication, such as fiscal quarters, for financial reporting requirements.**

Sure, it's easy enough to perform basic time period arithmetic, such as adding and subtracting days, comparing two `DateTime` objects for equality, or determining if one comes before or after the other. However, creating a time range has always been—at least for me—quite a laborious process.

Let's step through some code, so you'll understand what I mean visually. Let's say I need to do some financial reporting in my application, specifically by fiscal quarter. As a rough example, using PHP's DateTimeImmutable class[2], I could work out a set of ranges as you can see in Listing 1.

The code, as rough as it is, contains a single function, `getQuarter()`, which will return an array of two `DateTimeImmutable` objects, based on the function arguments supplied. Those function arguments are the year and the year's quarter that you want the range for.

You can see that it's not *too* involved; making use of a switch statement to build the quarter value, based on one of four acceptable options—naturally, there can be only four. When calculated, it then uses a bit of `sprintf` magic to build the string to initialize the `DateTimeImmutable` object for the start of the quarter.

Then, it creates a second one by calling the first one's `add()` method and supplying a `DateInterval` object with an interval of 90 days. Assuming those two objects can be instantiated, they're returned in an array. If for some reason, they cannot be, then an exception is thrown.

In a more fully thought out application, you could refactor the logic in any number of directions to initialize them more elegantly. However, it works.

Or, does it? If you were paying attention, you'd know there's a key bug in my code; the interval specification passed to the new DateInterval object. What

**Listing 1**

```
1. <?php
2.
3. /**
4.  * @param int $qtr
5.  * @param int $year
6.  * @return DateTimeImmutable[]
7.  * @throws Exception
8.  */
9. function getQuarter(int $qtr, int $year): array {
10.    if (in_array($qtr, [1, 2, 3, 4])) {
11.       switch ($qtr) {
12.          case (1): $quarter = 1;  break;
13.          case (2): $quarter = 4;  break;
14.          case (3): $quarter = 7;  break;
15.          case (4): $quarter = 10; break;
16.       }
17.       $q1Start = new DateTimeImmutable("{$year}-{$quarter}-01");
18.
19.       // add 90 days to get next quarter
20.       $q1End = $q1Start->add(new DateInterval('P90D'));
21.       return [$q1Start, $q1End];
22.    }
23.    throw new \Exception('Range requested does not make sense');
24. }
```

1 *PHP's DateTime library: http://php.net/class.datetime*

2 *PHP's DateTimeImmutable class: http://php.net/class.datetimeimmutable*

do you think it would print out if we were to run it, using the code below?

```
$period = getQuarter(1, 2017);
printf (
   "Starts: %s. Ends: %s",
   $period[0]->format('d.M.Y'),
   $period[1]->format('d.M.Y')
);
```

Well, since January and March have 31 days, and February has 28, except on a leap year, when it has 29—I should know this as I got married on February 29—then, it won't report the start and end of the quarter correctly. Instead, here's what it will print out:

```
Starts: 01.Jan.2017. Ends: 01.Apr.2017
```

The start date is fine, but note the end date of `01.Apr.2017`. That's the first day of the following quarter, not the current one. Given that, we'd have to add further logic or fiddle with how we add to our intervals to correctly calculate the quarters, determining the number of days within each.

**Listing 2**

```
<?php

class DateTimeRange
{
    const DAYS_IN_Q1 = 89;
    const DAYS_IN_Q2 = 90;
    const DAYS_IN_Q3 = 91;
    const DAYS_IN_Q4 = 91;

    /**
     * @param int $qtr
     * @param int $year
     * @return DateTimeImmutable[]
     * @throws Exception
     */
    public static function getQuarter(int $qtr, int $year): array {
        if (in_array($qtr, [1, 2, 3, 4])) {
            switch ($qtr) {
                case (1):
                    $quarter = 1;
                    $interval = self::DAYS_IN_Q1;
                    break;
                case (2):
                    $quarter = 4;
                    $interval = self::DAYS_IN_Q2;
                    break;
                case (3):
                    $quarter = 7;
                    $interval = self::DAYS_IN_Q3;
                    break;
                case (4):
                    $quarter = 10;
                    $interval = self::DAYS_IN_Q4;
                    break;
            }
            $q1Start = new DateTimeImmutable("{$year}-{$quarter}-01");
            $q1End = $q1Start->add(
                new DateInterval("P{$interval}D")
            );
            return [$q1Start, $q1End];
        }
        throw new \Exception('Range requested does not make sense');
    }
}
```

We could instead use constants to do this for us, avoiding the need to calculate anything. To help visualize what I mean, I did a little refactor of the original code, which you can see in Listing 2, that pulls the function into a class and makes it a static.

However, the point remains, it would start to get a bit more complicated. Moreover, that's just for the basic fiscal quarter calculations. Oh, and there's another potential mistake. Have you spotted it? Does quarter one in your country's fiscal year start on the first of January? If you look around the world, you'll see it varies in all kinds of ways.

*Instead of hard-coding the number of days in each quarter, you could also compute the end of a quarter by adding a period of three months and then subtracting a day but the point remains, its not a straight forward solution in any case.*

Stepping beyond this initial example, let's consider three follow-on questions:

- What if we wanted to calculate a date range based on an arbitrary month—allowing for leap years?
- What if we wanted to create one based on a year?
- What if we wanted to create one based on a semester?

That's a lot of code for an edge case we have to write (along with the requisite tests of course). If that were the case, we'd have to start creating a lot of code. Ideally, that's not something which we'd be keen on—especially if it's not the core logic that you're tasked with solving in your application by our client. Don't add to your technical debt, let's look for a third party library which handles this for us.

Then there's another problem that may arise, one you don't want to get into: the case of the class that knew too much, or the function that attempted to do too much. For example, if you had a

function which printed a report, that looked a little like the code below:

```
function generateReport($year, $quarter, $client, $format) {
    $qStart = new DateTimeImmutable(
      sprintf('%d-%d-01', $year, $quarter)
    );
    $qEnd = $qStart->add(new DateInterval("P90D"));

    // ... rest of the reporting code
}
```

Please excuse the fact I've tried to exaggerate how much it's breaking SRP[3], indulging in feature envy, by having the class want to know about the client and the report format—in addition to working out the date range. This is just to draw acute emphasis to the point I'm trying to make.

But, this kind of code isn't uncommon. In the past, I've written code like this, and I've come across numerous code bases where this is also the case. Now if we can assume this has been written at least once, then I'd suggest it's also fair to assume it's written in many other locations in our fictitious code base as well.

This is our justification for creating a reusable library which handles this for us, so we don't have the need to create one for ourselves. It's for these reasons that I'm going to spend the rest of the column introducing Period[4], a Time range API for PHP, maintained by The League of Extraordinary Packages[5]. To quote the package's website, here are the four key reasons why you want to get to know this package:

1. It treats a time range as an immutable value object.
2. It exposes many named constructors to ease time range creation.
3. It covers all basic manipulations related to time range
4. It's framework-agnostic.

In short, by using it, we can offload a lot of work by using a package that provides a well thought out, unified interface for time range creation. Also, we can extract the creation of time ranges to a single library.

Sound good?

Then let's dig a bit deeper, and see what it has to offer. Here's a quick summary, before we dive into some code examples:

- Time ranges can be created based on *day*, *duration*, *month*, *quarter*, *semester*, *week*, *year*, *year interval*, and *date points*.
- Time ranges can be *added*, *intersected*, *subtracted*, *validated*, *split*, and *compared*.
- We can check if one has a longer or shorter duration than another, diff one against another, check if one contains another, or check if one contains a gap.
- And, quite a bit more.

## Installation

It's hardly worth mentioning, as I say this almost every month. However, first we have to install Period and to do that, we are going to use Composer. From the terminal, in the root directory of your project, run:

```
composer require league/period
```

When that's done, we're ready to get to work.

## Creating a Time Range for Quarters One Through Four in 2017

Let's start off by creating a time range for the four quarters we created code for previously. To do that, here's the massive amount of code required:

```
$qOne2017 = Period::createFromQuarter(2017, 1);
$qTwo2017 = Period::createFromQuarter(2017, 2);
$qThree2017 = Period::createFromQuarter(2017, 3);
$qFour2017 = Period::createFromQuarter(2017, 4);
```

A `var_dump` to inpect one of these looks like Output 1.

**Output 1**

```
object(League\Period\Period)#15 (2) {
  ["startDate":protected]=>
  object(DateTimeImmutable)#13 (3) {
    ["date"]=>
    string(26) "2017-10-01 00:00:00.000000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(3) "UTC"
  }
  ["endDate":protected]=>
  object(DateTimeImmutable)#17 (3) {
    ["date"]=>
    string(26) "2018-01-01 00:00:00.000000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(3) "UTC"
  }
}
```

In four lines of code, we now have four Period objects modeling quarters one through four of this year. If we were to print out any one of the objects, e.g., `print $quarterFour2017;`, as Period implements the magic `__toString()` method, then it would print out a string representation of the time

3 *SRP:* http://phpa.me/wikipedia-srp

4 *Period:* http://period.thephpleague.com

5 *The League of Extraordinary Packages:* http://thephpleague.com

range. In the case of `$quarterFour2017`, it would print out `2017-10-01T00:00:00Z/2018-01-01T00:00:00Z`. You can see that it prints out UTC representations of the start and end dates in ISO 8601 format.

What if we just wanted to know either the start or end date? No problem! There are accompanying functions, which return `DateTimeImmutable` objects. Here's an example of using the `getStartDate()` method, and then calling the returned `DateTimeImmutable` object's format method to print an ISO 8601 representation of the start date.

```
print $qFour2017->getStartDate()->format(DateTime::ISO8601);
```

Let's get a bit more complex and look at determining the intersection of two time ranges. Have a look at the code below. Here, we're first creating a time range for April 2017 (`$april2017`). Then, we're creating a time range for week 15 in 2017 (`$week15`), which is the second week of April this year. Next, we create another Period object, which is the time range where the two intersect, which is week 15.

```
$april2017 = Period::createFromMonth(2017, 4);
$week15 = Period::createFromWeek(2017, 15);
$intersection = $april2017->intersect($week15);
```

By calling the code below, we can see the start and end dates, in a simpler format.

```
printf(
    "Start: %s / End: %s",
    $intersection->getStartDate()->format('Y-m-d'),
    $intersection->getEndDate()->format('Y-m-d')
);
// Prints out: "Start: 2017-04-10 / End: 2017-04-17"
```

## A Little Time Manipulation

No, I'm not Doctor Who, but time manipulation is something I always wanted to do more easily with DateTimeImmutable, but always found convoluted. Luckily, it's a lot simpler in Period, specifically, let's see how to move both forward and backward from the current Period, using the same time scope. What's extra cool about Period is that regardless of how you created the Period object, you can use the same code to move either forward or backward in time.

Say that we create a Period object, for quarter four of this year, as in the following example:

```
$qFour = Period::createFromQuarter(2017, 4);
```

Now, we want to move forward to the next quarter, quarter one, 2018. To do that, we'd make use of two Period functions: `getDateInterval()` and `next()`, as follows:

```
$qNext = $qFour->next($qFour->getDateInterval());
```

To see when the next quarter starts:

```
print $qNext->getStartDate()->format(DateTime::ISO8601);
// 2018-01-01T00:00:00+0000
```

Both `next()` and its complement `previous()` take a `DateInterval` object to determine where to move either forward or back to. Given that, we can supply one by passing the value of the current Period object, by calling its `getDateInterval()` method. Becuase that method is available, irrespective of whether we created the Period object based on a *week*, *semester*, *quarter*, *year*, or *specific range*, we can now use a consistent interface, with minimal code, to move forwards and backward in time. There's no fiddly, tricky code to remember, based on the type of time range we've created

## In Conclusion

From the use case perspective, as well as the examples presented, you can see just how handy this library is. Take a look at the online documentation to unlock what else it can do. It's one I'm keen to make much more use of in the future, and one I want to wholeheartedly thank Oscar Merida for introducing me to it. Have a play around with Period and tweet me your thoughts on it—I'm @settermjd[6].

*Matthew Setter is an independent software developer and technical writer, focusing on security, continuous development, and Zend Expressive. He's currently writing a new book: Zend Expressive Essentials, which teaches the fundamentals of building applications with Zend Expressive. Share your email address if you're keen to know when it's ready!*

6 *@settermjd: https://twitter.com/settermjd*


# 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**