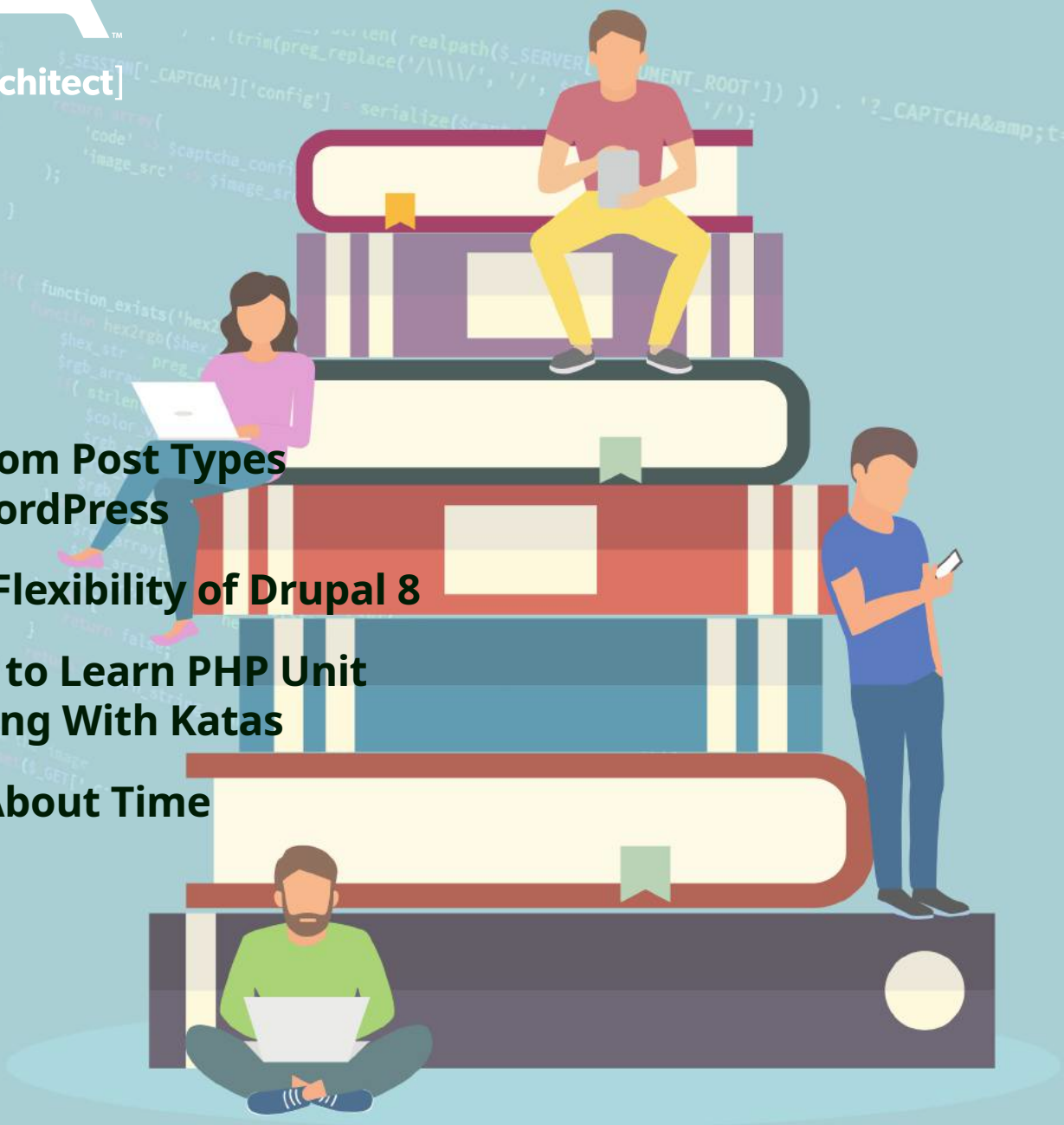
www.phparch.com

December 2018
Volume 17 - Issue 12

php[architect]

Better Practice



Custom Post Types in WordPress

The Flexibility of Drupal 8

How to Learn PHP Unit Testing With Katas

It's About Time

Free Sample Article

ALSO INSIDE

PHP[TEK] 2019
Conference

Save the Date!

The only conference that provides
deep-dive, sequential sessions for senior developers and
entry-level, need-to-know topics for beginners.

A conference experience tailored to meet your learning needs.

tek.phparch.com

CONFERENCE

Loudermilk Conference Center
May 21-23, 2019
Atlanta, GA

It's About Time

Colin DeCarlo

As applications scale and gain adoption, dates and time become much more of a concern than they once were. Bugs crop up, and developers start learning the woes of time zones and daylight saving time. Why did that reminder get sent a day early? How could that comment have been made at 5:30 a.m. if the post didn't get published until 9:00 a.m.? Indiana has how man time zones?!

Luckily, PHP developers have the tools they need to face these problems head-on and take back control of their apps.

Introduction

Time is a complicated concept. I'd even go as far as saying it's pretty much impossible to understand completely. Luckily though, we don't need a complete understanding of time to get along, we just make it up as we go. It's a strange moment when you realize time is both constant and completely arbitrary all at the same…time. What's constant is, as the saying goes, "Time waits for no man," it just goes and goes indefinitely. What's arbitrary is how we choose to measure it. For instance, have you ever wondered why there are seven days in the week?

Pushing the idea even further, have you ever wondered why we even measure time? What's the purpose? We measure time so we can record it, specifically relative to significant events. So what would you say is the most significant event? Likely, the beginning of time itself, and when was that? It depends on whom you ask.

The term "epoch" represents an instant in time chosen as a beginning, or reference point, for the measurement of the passage of time. In computing, many epochs exist[1] but in the web world, only one is used as our anchor point, the Unix epoch. The Unix epoch date is midnight, January 1st, 1970 Coordinated Universal Time (UTC). It's this measurement of time PHP uses as it's own basis for time-related functionality.

> *What's interesting is the Unix epoch is translated and expressed in terms of a different epoch. Nothing about time is straightforward*

Down to Basics

The most primitive time-related function in PHP is `time`. `time` returns the number of seconds which have passed between the epoch and the moment the function is invoked. The return value is referred to as *epoch seconds* or a *timestamp*. Epoch seconds, on their own, aren't handy. There isn't much you should do with them, though there are many things that you could do with them. Many developers use epoch seconds to perform date arithmetic, time zone conversions, and numerous other functions. Thankfully, PHP ships with a set of classes which provide an abstraction over `time` which proves to be much more useful and less error prone than using `time` alone.

However, `time` isn't the only function in PHP which returns epoch seconds, there's also `strtotime`. This lovely function is enjoyable to use. What makes it so much fun to toy around with is `strtotime` translates most English descriptions of dates and times and convert them into epoch seconds with remarkable effectiveness. For instance, `strtotime` correctly interprets mundane date strings like `"2018-10-06"` but also wildly relative ones too like `"first Saturday of next month"` and even `"first Saturday of next month plus one month"`. Unfortunately, returning epoch seconds wastes all this useful functionality, since they don't have much use on their own. Not all is lost though, PHP's `DateTime` class wraps much of the functionality of PHP's date and time functions, for instance, we can construct a `DateTime` object with the same descriptions of time that `strtotime` accepts. `DateTime` objects can also be modified with a similar syntax.

Moving up a rung in the ladder of usefulness is the `date` function. `date` accepts up to two arguments, a format string and, optionally, an integer representing epoch seconds. As it's return value, `date` generates a date/time string formatted according to the provided format string for the provided timestamp relative to the current time zone. Note that if no second argument is supplied, `date` uses the current date and time as the timestamp. `date` can be useful in the presentation layer of modern PHP applications as a pragmatic way to display the copyright year.

The `date` function isn't alone in its responsibility to provide formatted, human-readable date strings. It has a companion function called `gmdate`. When used, `gmdate`, as you may have inferred, returns a formatted date string relative to Greenwich Mean Time (GMT). GMT is a special time zone. Originating from the Royal Observatory in Greenwich, London, it is considered the basis for all other time zones in the world. It has no offset from UTC and does not observe daylight saving time. These characteristics make GMT, and transitively `gmdate` a particularly useful time zone to work with. Resultant times are consistent and convert easily to other time zones.

1 *epochs exist:* https://phpa.me/wikip-notable-epochs

The DateTime Object

When working with time-based data in your application, you'll want to stay away from using the basic date and time functions for all but the most straightforward cases. For instance, tagging a filename with a timestamp or perhaps displaying the current date in a formatted string. For all other scenarios, use instances of PHP's `DateTime` class because manipulating time-based data is difficult and error-prone. Time has a lot of edge cases (especially around time zones) and adding and subtracting a precise number of seconds from a timestamp will not always yield the desired result.

Constructing DateTime Objects

There are two direct ways to construct `DateTime` objects, namely using the `DateTime` constructor itself with `new` or using the `createFromFormat` static method. The `DateTime` constructor is very versatile as it uses the same parser as `strtotime` to understand it's input, therefore, you can use strings such as "now", "May 11th 2009", "last day of next month" and "2015-02-19 13:45:00" to create your objects.

```
$tomorrow = new \DateTime("+1 day");
```

Additionally, the `DateTime` constructor accepts a Unix timestamp prepended with an `@` symbol (e.g. `@1483142400`). This call initializes the `DateTime` object to the time specified by the timestamp in the UTC time zone.

```
$tomorrow = new \DateTime("@0");
```

DateTime Constructor Gotchas

Sometimes you'll get unexpected results from constructing `DateTime` objects.

When specifying dates without a definite time portion the current time is sometimes used (and sometimes not). For instance, strings such as `"today"`, `"yesterday"`, `"tomorrow"`, `"last monday of next month"`, `"March 31, 1981"`, and `"2007-06-08"` will result in the time portion of the `DateTime` object to be initialized to `00:00:00`. However, strings like `"next week"`, `"last Wednesday"`, `"last day of next month"`, and `"+1 week"` initialize the time portion to the current time.

Some date formats are ambiguous, for instance `"10-11-12"` could represent October 11th, 2012, November 10th, 201, or many other possibilities. In instances such as this, PHP's `strtotime` parser is trained to use the separator as a hint regarding how to parse the date. This is due to different regions using different separators; the order of year, month, and day in the date string can be guessed based on the customs of the region.

In instances where we can interpret the time zone from the date string, the interpreted time zone overrides any supplied or system default time zone.

When the format of the date string is known, it is preferable to use the `DateTime::createFromFormat` named constructor. This is especially the case if the incoming date strings have the potential to be ambiguous. It is important to remember any portion of the date and time not specified by the input format are initialized to their current value. That is, if the current time is `08:45:15`, and the format string being used to create the `DateTime` object is `Y-m-d`, then the string `2009-05-11` creates a `DateTime` object with value `2009-05-11 08:45:15`. It is possible to override that behavior, though, by using either the `!` or `|` format modifiers. `!` resets any fields preceding the `!` to the Unix epoch.

```
DateTime::createFromFormat(
  'Y-m-!d H:i:s', '2017-02-04 01:23:45'
);
// '1970-01-04 01:23:45
```

`|` resets any fields not parsed to the Unix epoch.

```
DateTime::createFromFormat(
   'Y-m-d|', '2017-02-04'
);
// '1970-01-04 00:00:00'
```

Time Zones

Both the `DateTime` constructor and `createFromFormat` static method accept an optional `DateTimeZone` as their final parameter. This object is used to identify the time zone the resultant `DateTime` is relative to. Note, however, this optional parameter may be ignored if the time zone of the date string used to construct the object can be determined. For instance, if the `DateTime` is constructed using a timestamp, the time zone of the `DateTime` instance is always `UTC`. This behavior is a common gotcha for developers when they create the `DateTime` object using a timestamp (perhaps the `created_at` value of a database record) passing in a `DateTimeZone` instance relative to the users own time zone. The poor developer suspects that when the `DateTime` object is formatted as a string—which appears to be in the user's time zone—only to find out they are hours off the mark. Much confusion can ensue! In scenarios such as this, the time zone can only be changed after the object is constructed using the `setTimezone` method.

The importance of using a `DateTimeZone` instance when constructing `DateTime` objects with any non-relative string values (e.g., `1982-08-18 04:13:12`) cannot be understated. The reason is strings such as this contain no time zone information, as such, the PHP runtime's default time zone setting is used and may lead to hard to track bugs if the default time zone is not the intended time zone.

To mitigate the risks of constructing `DateTime` objects relative to unintended time zones, you should ensure PHP's default time zone is set to a known value and not rely solely on a preset value. Do this by either setting the `date.timezone` directive in `php.ini` to a specified, supported, time zone name. If, however, you don't have access to the `php.ini` file, you can use the `date_default_timezone_set` function early in your application bootstrap to specify the time zone you wish all dates and times to be constructed relative to.

```
# php.ini
date.timzone="UTC"
```

```
// bootstrap.php
date_default_timezone_set('UTC');
```

Constructing DateTimeZone

The `DateTimeZone` constructor accepts a single string parameter used to identify the time zone the object represents. This string can be either a supported time zone name[2] or a string identifying the number of hours and minutes offsetting the time zone from UTC. (e.g. `+0100` or `-0400`).

In applications which support users in multiple time zones, it's common to store a user's time zone in their user record. These time zones should be stored as one of the PHP supported time zones and not as an offset. Time zone names and offsets cannot be used interchangeably as some time zones observe daylight saving time during the summer months which temporarily change their offset by one hour. As such, if you were to store each user's time zone as an offset from UTC (either in the standard `+/-HHMM` format or possibly even the number of seconds), users in time zones which do observe daylight saving time would have their dates and times misreported for nearly eight entire months! Using a supported time zone, on the other hand, ensures correct reporting year round.

Time Zones And MySQL

MySQL supports three individual time zone settings, a `system` time zone setting, a `global` (or server) time zone setting, and a `client` time zone setting. The `system` time zone is set when the MySQL server starts and can not be changed. The `global` time zone setting has a default value of `SYSTEM` (which references the `system` time zone) but may be changed to something different either at start up or by a user with the appropriate privileges. The `client` time zone setting is set on each connection to the server; if not provided, the `global` time zone setting is used.

MySQL uses the time zone setting in time functions like `NOW` and `CURDATE`. More importantly, though, it is also used when storing values into or retrieving values from `TIMESTAMP` columns. Internally, MySQL stores all `TIMESTAMP` values as epoch seconds, so a conversion is always performed from the current time zone setting to UTC when storing data and from UTC to the current time zone setting when retrieving data.

Aligning your time zone settings between PHP and MySQL is necessary. If you aren't careful, it will result in inconsistent data which could lead to very difficult to track bugs.

For example, consider the following scenario:

The default PHP time zone is set to `America/Toronto` and your application is querying a MySQL server configured to use the `UTC` time zone. Your application is an e-commerce platform and records every individual sale into a `sales` database table.

When a sale is made, the sale details are saved to the database with a query like:

```
INSERT INTO `SALES`
   (`item`, `purchase_amount`, `purchased_at`)
VALUES
   ('Watchamacallit', '1500', '2018-09-01 21:32:16')
```

At the time, your application believes this item was purchased just after 9:32 p.m. on September 1st, 2018. However, MySQL has also interpreted this sale as happening just after 9:32 p.m. on September 1st, 2018 *in the UTC time zone* which is four hours ahead of `America/Toronto`. Because we're not storing the timezone explicitly, the specific time can be interpreted by both PHP and MySQL which could have their assumption about the associated time zone.

This bug can go unnoticed indefinitely as long as the time zone settings of both the PHP runtime and the MySQL server never change. The PHP application can continue to happily report dates and times relative to `America/Toronto` and the MySQL server can continue to interpret them as relative to `UTC`. Shenanigans!

Never Use a Time Zone Which Observes DST

Staying with the above scenario, avoid using a default time zone setting which observes daylight saving. This is simply due to the rules of daylight saving where once a year one day has 2:00 a.m. twice and another day has no 2:00 a.m. at all.

The fallout from using a default time zone which observes daylight saving can be quite severe if automated tasks are set to execute at 2:00 a.m. 2:00 a.m. may seem like an obscure time to choose to run tasks at, but it's quite easy for this scenario to arise. Consider the following.

Jane is a developer on the project and is asked to run daily database backups. She references the server logs and identifies there isn't much load on the system in the early hours of the day. Since this is a daily backup, it then makes sense to schedule it for 12:00 a.m.

John is a data analyst on the project and needs to update the Data Warehouse daily as well. He writes a script to perform the update and sees the backup happens at 12:00 a.m. Not knowing just how long the backup takes, he assumes it couldn't possibly take more than an hour and schedules his update script to run at 1:00 a.m.

Finally, Mark, another developer on the project, needs to ensure subscribers are charged for access to the application on a monthly basis. He writes a script to identify users who need to be charged based on the start of their subscription and would like to schedule it to run daily. He sees the database backups run at 12:00 a.m. and the Data Warehouse is updated at 1:00 a.m. Mark then assumes the pattern is to execute automated scripts at single hour intervals and schedules his subscription script to run at 2:00 a.m.

Months later in early November, many complaints come into customer service reporting users have been charged

2 *time zone name:* http://php.net/timezones

Listing 1

```
1. $all = DateTiemZone::listIdentifiers();
2.
3. $includingDeprecated = DateTimeZone::listIdentifiers(
4.    DateTimeZone::ALL_WITH_BC
5. );
6.
7. $europeAndAsianRegions = DateTimeZone::listIdentifiers(
8.    DateTimeZone::EUROPE | DateTimeZone::ASIA
9. );
10.
11. $excludingAntarctica = DateTimeZone::listIdentifiers(
12.    DateTimeZone::ALL ^ DateTimeZone::ANTARCTICA
13. )
14.
15. $justCanada = DateTimeZone::listIdentifiers(
16.    DateTimeZone::PER_COUNTRY, 'CA'
17. );
```

twice for their monthly subscription. As well, many customers have directly contacted their credit card providers and identified the second charge as fraudulent. As a result, customer service agents spend many hours handling calls from angry subscribers, months of subscription revenue is lost by agents comping accounts to win back good favor and the company's reputation with its creditors is tarnished leading to harder to negotiate contracts.

The development team learns a valuable and costly lesson: daylight saving time is the worst.

Listing All Available Time Zones

It's possible, as in Listing 1, to generate a list of all available time zone names using the `listIdentifiers` static method on `DateTimeZone`. The method accepts two optional parameters which can be used to filter the listing based on time zone region or country.

It's possible (but difficult) to use this method to create an exhaustive yet unintimidating time zone selection menu. However, you should try to avoid such selection menus altogether. No matter how you slice it down, there will always be many time zones in the list, and people get confused easily. There are many services which can geolocate[3] based on IP address or other provided information such as an address. If your application must support users in different time zones, it's best to try to detect their time zone and allow them to change it if they need to.

Date Arithmetic

Math with dates can be tricky because the way we measure time itself isn't absolute. For instance, in most circumstances, a day lasts 86,400 seconds, with the exception, of course, for days in time zones which observe daylight saving. In those time zones, one day has 90,000 seconds, and another has 82,800.

Luckily, it all comes out in the wash by the end of the year, and we can safely say a year has 31,536,000 seconds, except when it doesn't. Some years are leap years where we need to add a day to the year to align our calendars with the position of the earth relative to the sun as we've let it drift a little for the past three years. Of course, leap years aren't every four years, that would be ridiculous. Should we mention leap seconds?

> *In the Gregorian calendar three criteria must be taken into account to identify leap years: The year can be evenly divided by 4; If the year can be evenly divided by 100, it is NOT a leap year, unless; The year is also evenly divisible by 400. Then it is a leap year.*

Bringing time back down to earth, a month isn't even a consistent number of days; they either have 30 or 31 days except, of course, one month that has only 28 days except in some years where it has 29. Time is constant and out of our control, but our measurement of it is entirely within our control and full of exceptions. Entire days have been wiped from the calendar as if they never happened; no one was born or died between October 4, 1582 and October 15, 1582, because those days *never existed*. They were removed from the calendar by Pope Gregory XIII in the papal bull *Inter gravissimas* to realign the vernal equinoxes.

> *Dates and calendars have confused people long before computers arrived on the scene. The 1908 Olympics were help in London, which used the Gregorian calendar. The Russia delegation set to participate in the shooting competition missed their event since they were still on the Julian calendar. The athletes showed up twelve days too late.*

All of this is to say, when doing date arithmetic, be careful and don't add seconds to timestamps. Instead, you can use a variety of methods which are available on the `DateTime` instance.

Adding And Subtracting from Dates

If you want to add or subtract some amount of time to or from a `DateTime` instance, you can use either the `add` or `sub` methods, respectively. Each method accepts as it's argument, a `DateInterval` representing the amount of time you wish to add or remove.

A `DateInterval` can be constructed using either the class' constructor or by using the static method `createFromDateString`.

Constructing with the Constructor

The `DateInterval` constructor accepts a string which adheres to the ISO8601 duration specification[4]. This specification offers a very concise and exact

3 *geolocate:* https://www.maxmind.com/

4 *ISO8601 duration specification:* https://phpa.me/wikip-iso8601-durations

way to represent an amount of time. An ISO8601 duration takes follwoing form.

```
P[n]Y[n]M[n]DT[n]H[n]M[n]S
```

For example, `P1Y3M26DT4H10M59S` specifies a duration of "One year, three months, twenty-six days, four hours, ten minutes, and fifty-nine seconds." Any portion of the specification can be omitted (except for the `P`) if it is not required (e.g., `P3M` for "three months") and any portion can exceed its "standard" limit (e.g., `PT48H` for "forty-eight hours").

Creating with `createFromDateString`

As an alternative, a `DateInterval` can be created using a friendlier notation via the `createFromDateString` method. This notation is certainly more verbose but much easier to reason about for simple intervals. An interval of two weeks can be created by passing the string `"2 weeks"` to this method. Multiple portions of the interval can be specified as well, `"3 months + 6 days"` creates an interval lasting three months and six days.

An alternative method which you can use to modify `DateTime` instances is the `modify` method. This method accepts strings which look remarkably similar to the strings accepted by the `DateInterval::createFromDateString` method. Almost as if under the covers, a `DateInterval` is created from it and passed into the `add` method, but it's probably just a coincidence.

A common gotcha and source of nasty bugs is that the `DateTime` object is mutable. Meaning any time you use the `add`, `sub`, or `modify` methods the object value is changed. Add this behavior to the fact objects are passed by reference in PHP, and you have a recipe for disaster. Any time you hand a `DateTime` object over to an outside function you are relinquishing control over it, helpless to the possibility the function may change its value. Thankfully, though, PHP 5.5 introduced the `DateTimeImmutable`[5] class which:

> *behaves the same as DateTime except it never modifies itself but returns a new object instead.*

Listing 2

```
1. $utc = new DateTime(
2.     '2006-09-30 04:00:00',
3.     new DateTimeZone('UTC')
4. );
5. $toronto = new DateTime(
6.     '2006-09-30 00:00:00',
7.     new DateTimeZone('America/Toronto')
8. );
9.
10. $x = ($utc == $toronto); // true
```

5 `DateTimeImmutable`: *http://php.net/class.datetimeimmutable*

Using `DateTimeImmutable` in place of `DateTime` protects your application from this sort of bug at virtually no cost.

Comparisons

`DateTime` objects can be directly compared for equality much the same as primitive values (see Listing 2). The relative time zone of the `DateTimes` being compared is not considered during the comparison. An excellent way to think about the comparison is the objects are compared with respect to *when they are in time* and not *where* they are.

Retrieving the Difference Between DateTime Objects

In addition to being able to compare two `DateTime` objects, it can sometimes be useful to know the difference in time between the two objects as well. The `DateTime` class offers the `diff` method for these purposes. Passing a `DateTime` object into the `diff` method returns a `DateInterval` object representing the difference in time between the two. The `DateInterval` object has public attributes to specify the number of years, months, days, hours, minutes, and seconds the two `DateTimes` are from each other.

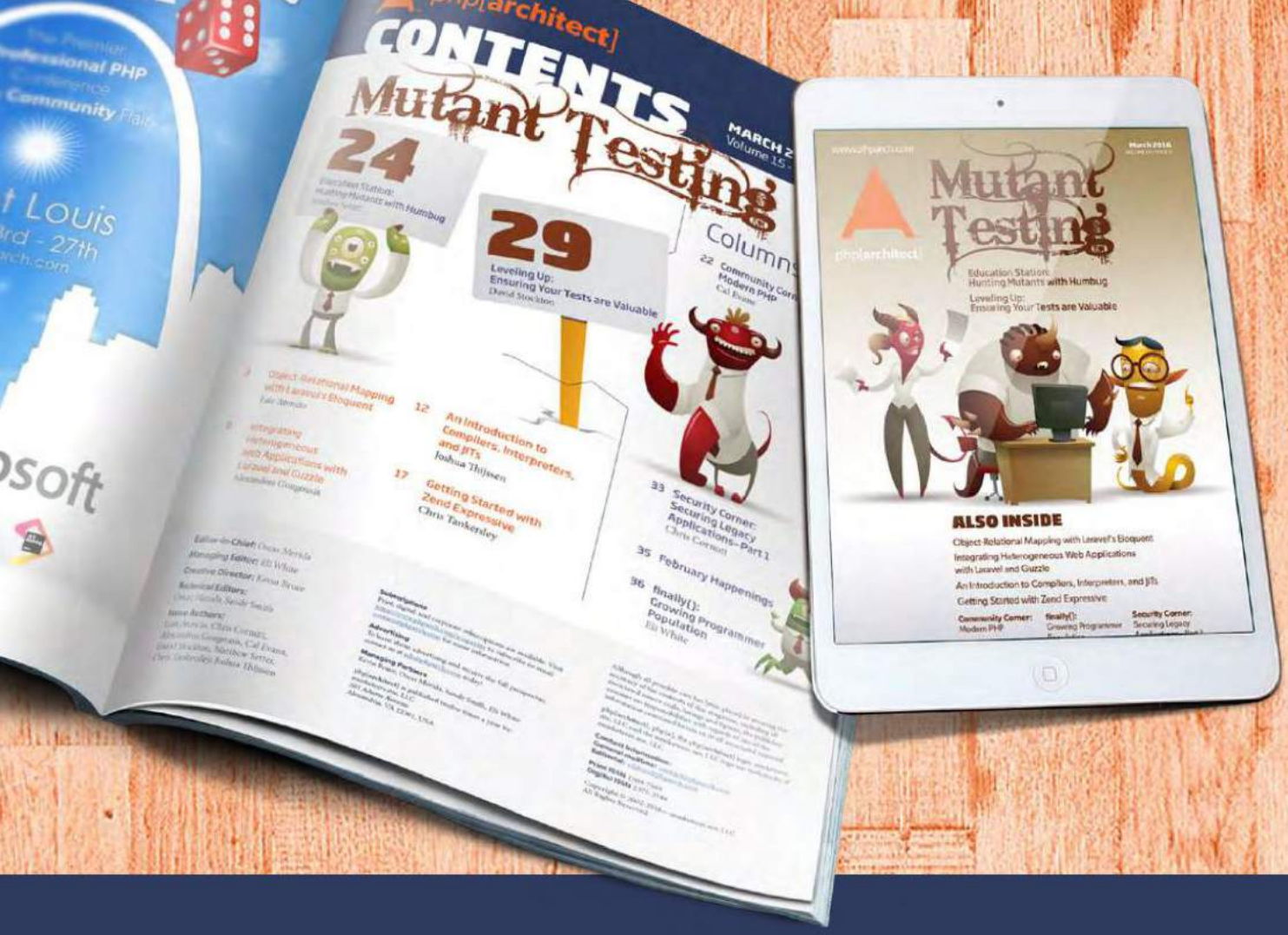
Closing Time

Just as the passage of time is inevitable, so too is the moment when a developer has to come face to face with managing and manipulating time. It's important at that moment to remember many developers before you have also encountered these exact issues. Proof of this is the existence of PHP DateTime library and supporting functions. Using these functions allows you to manage time effectively and efficiently *and* with confidence.

Colin is a senior developer with Vehikl, a full services software consultancy, in Waterloo, Ontario, Canada and co-organizer of the GPUG PHP Users Group. With over a decade of professional experience, Colin is never put off by a good challenge. He enjoys refactoring often neglected legacy code bases as much as tackling the wide open space of green field projects. No matter the context, he puts great effort into delivering simple solutions with as little code as possible. @colindecarlo

Related Reading

- *Education Station: Simple, Compact Time Range Creation with Period* by Matthew Setter. July 2017. https://www.phparch.com/magazine/2017-2/july/
- *Education Station: Date and Time Handling with Carbon* by Matthew Setter. January 2016. https://www.phparch.com/magazine/2016-2/january/
- *Leveling Up: Getting a Date with PHP* by David Stockton. May 2015. https://www.phparch.com/magazine/2015-2/may/



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.

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

http://phpa.me/mag_subscribe

php[**architect**]