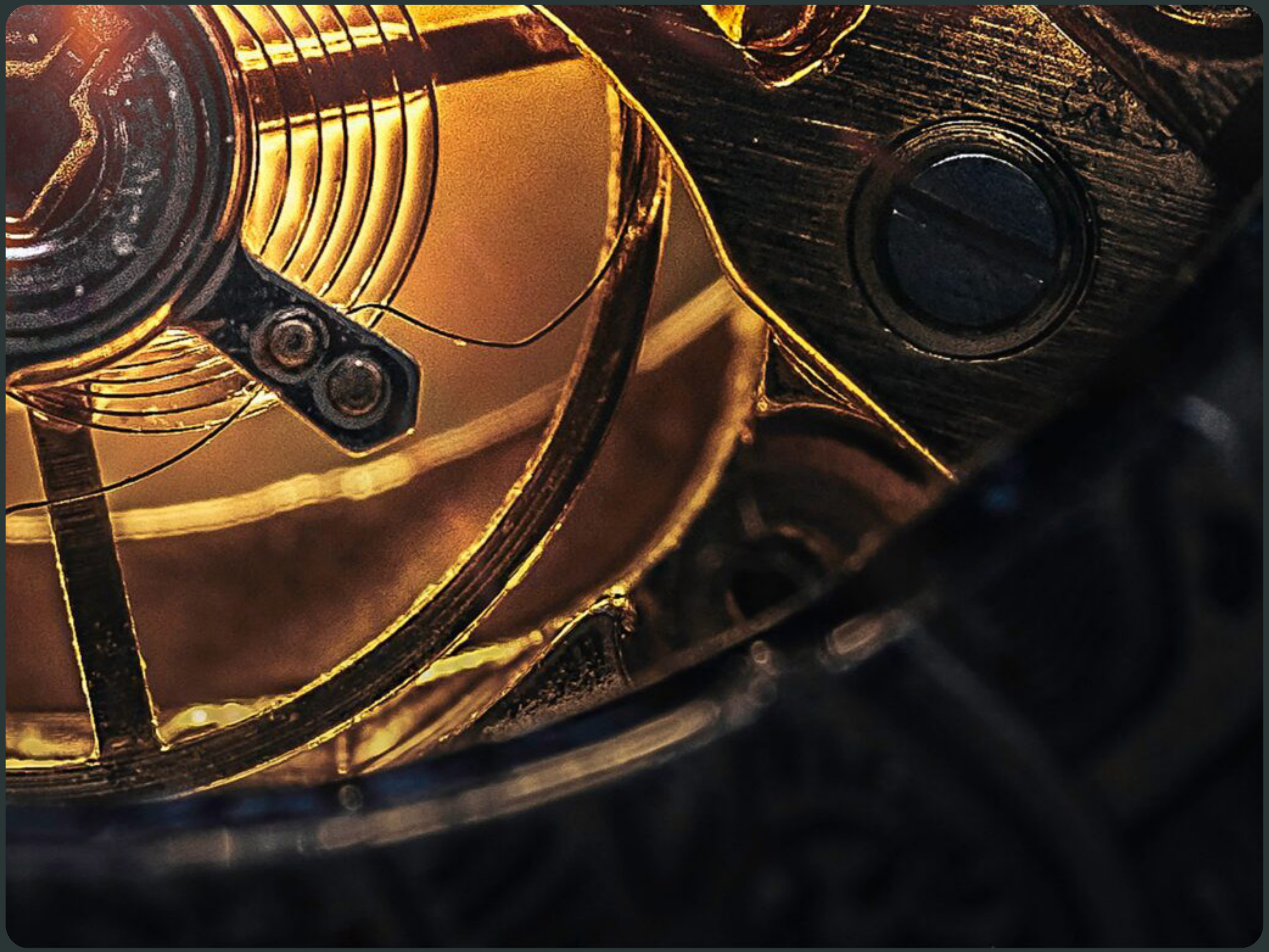
# Working with Time Zones, Timestamps and Datetimes in Laravel and MySQL

This article clarifies the complexities of timestamps, datetimes, and time zones in MySQL and Laravel, offering best practices for handling dates and time zones effectively.

13 OCT, 2023 — 6 MIN READ



By Mats-Joonas Kulla
Co-founder, CTO

There seems to be quite a bit of confusion around how timestamps, datetimes and time zones really work. This article aims to demystify these concepts and give some

CATEGORY

eSignatures

IN THIS ARTICLE

SHARE

**Live product demo**

Book a 30-minute call with our expert to see how we can help your business.

Book a call

recommendations and best practices on how to handle dates and time zones in a sane way in your Laravel app and MySQL.

## How the TIMESTAMP Type Works in MySQL

The official documentation of MySQL explains it as follows:

> MySQL converts TIMESTAMP values from the current time zone to UTC for storage, and back from UTC to the current time zone for retrieval. (This does not occur for other types such as DATETIME.) By default, the current time zone for each connection is the server's time. The time zone can be set on a per-connection basis. As long as the time zone setting remains constant, you get back the same value you store. If you store a TIMESTAMP value, and then change the time zone and retrieve the value, the retrieved value is different from the value you stored. This occurs because the same time zone was not used for conversion in both directions. The current time zone is available as the value of the time_zone system variable. For more information, see Section 5.1.15, “MySQL Server Time Zone Support”.

This explanation is perhaps a little bit abstract. So, let’s add some context and see what is really happening behind the scenes.

## Current Time Zone

In order to understand how the timestamp conversions work we first need to know what's meant by **current time zone.**

In short, **current time zone** is the value of the SESSION time_zone. By default this is the SYSTEM time of the server that the database is running on. Let's run some queries to illustrate this.

Running `SELECT @@SESSION.time_zone;` will return the current SESSSION time_zone like this:

```
+---------------------+
| @@SESSION.time_zone |
+---------------------+
| SYSTEM              |
+---------------------+
```

Let's change the SESSION time_zone to "+02:00" by running: `SET SESSION time_zone = '+02: 00';`

`SELECT @@SESSION.time_zone;` will now return:

```
+---------------------+
| @@SESSION.time_zone |
+---------------------+
| +02: 00             |
+---------------------+
```

## Practical Examples of How Timestamp Works

Let's now go through some examples with specific dates and times to see how the timestamp storage and retrieval works in real life.

We'll start by creating a table with a TIMESTAMP column to store our test data.

```
CREATE TABLE timestamp_test (
   `timestamp` TIMESTAMP,
);
```

We'll now set our session time_zone to +02:00 and store some data

```
SET SESSION time_zone = '+02: 00';

INSERT INTO timestamp_test VALUES ('1970-01-01 03: 00: 00');
```

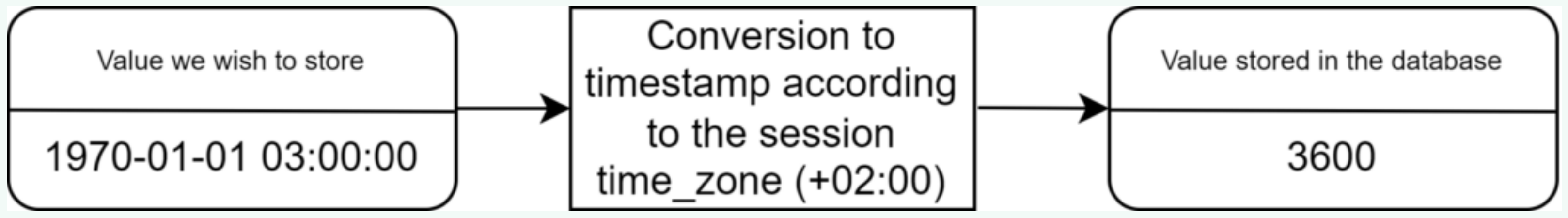
Check that the value got stored:

```
SELECT * FROM timestamp_test;
```

We'll see:

```
+---------------------+
| timestamp           |
+---------------------+
| 1970-01-01 03: 00: 00 |
+---------------------+
```

Schematic representation of the storage process:


Value we wish to store
1970-01-01 03:00:00
Conversion to timestamp according to the session time_zone (+02:00)
Value stored in the database
3600

So far so good. But what happens if we change the session time_zone?

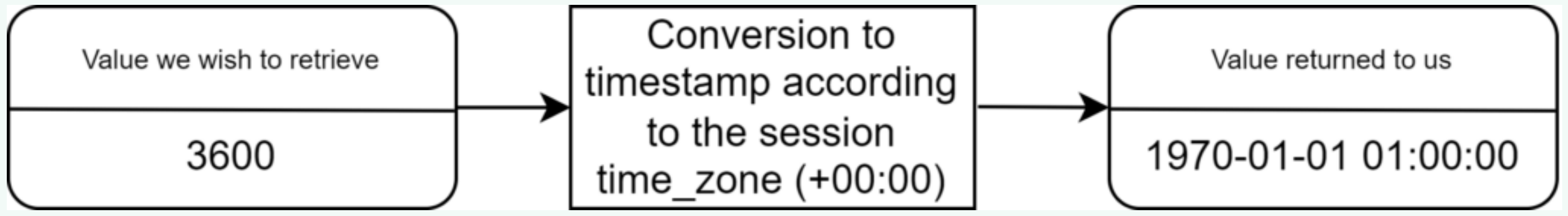
Let's set our session time_zone to `+00: 00` and retrieve the data again.

```
SET SESSION time_zone = '+00:00';
SELECT * FROM timestamp_test;
```

We'll see:

```
+---------------------+
| timestamp           |
+---------------------+
| 1970-01-01 01:00:00 |
+---------------------+
```

Schematic representation of the retrieval process:


Value we wish to retrieve
3600
Conversion to timestamp according to the session time_zone (+00:00)
Value returned to us
1970-01-01 01:00:00

**Key takeaways:**

- MySQL stores the timestamp value as a **Unix timestamp** in seconds.
- MySQL does **not** store any information about the timezone.
- Every time you store a value as a timestamp, it is converted to the Unix timestamp according to the current session time_zone.
- Every time you retrieve a timestamp, it is converted to the datetime value according to the current session time_zone.

**Note**: A simple algorithm to convert dates to timestamps according to a specific timezone might look something like this (in case you're interested how that's actually done):

- Get the difference between the datetime and the Unix epoch (1970-01-01 00:00:00) in seconds.

- Convert your current timezone offset to seconds
- Subtract your current timezone offset from the value you got in step 1.

For example, let’s say our time zone offset is +02:00 and we wish to convert 1970-01-01 03:00:00 to a Unix timestamp.

- 1970-01-01 03:00:00 - 1970-01-01 00:00:00 = 3h = 3 * 60 * 60 = 10800
- +02:00 in seconds is: 2 * 60 * 60 = 7200
- 10800 - 7200 = 3600

Another example, let’s say our time zone offset is -03:00 and we wish to convert 1970-01-01 08:00:00 to a Unix timestamp.

- 1970-01-01 08:00:00 - 1970-01-01 00:00:00 = 8h = 8 * 60 * 60 = 28800
- -03:00 in seconds is: -3 * 60 * 60 = -10800
- 28800 - - 10800 = 39600 (note that we actually add the values as double minuses give a +)

## How the TIMESTAMP type differs from the DATE and DATETIME types

In case of TIMESTAMP, the actual value that is stored and retrieved depends on the session time_zone whereas DATE and DATETIME are always retrieved as the exact same values that were stored. You can imagine the DATE and DATETIME values as static strings. The string you store does not change upon retrieval. You’ll always get back the exact same value that you stored no matter the database’s or session’s time zone.

TIMESTAMP can only store values from 1970-01-01 00:00:00 to 2038-01-19 03:14:17. The reason for this is how the Unix time is encoded: **https://en.wikipedia.org/wiki/Year_2038_problem**

DATETIME and DATE do not have such a limit.

## How Laravel Handles Dates and Times

We've seen how the timestamp works on MySQL's side. Let's now see how dates and times are handled by Laravel.

Laravel uses **Carbon** for generating dates (**https://laravel.com/docs/10.x/helpers#dates**). Carbon in turn uses PHPs Date/Time functions **https://www.php.net/manual/en/ref.datetime.php**. This means that when we generate a current date then that's done according to PHP's timezone. But what determines PHP's timezone? Well, Laravel conveniently does that for you via the config/app.php timezone setting.

What kind of implications does the above have on how the dates are saved to our database? We can bring an example to illustrate this.

Let's consider the following situation:

- timezone in our app's config/app.php is set to `Europe/Berlin`
- our database session time_zone is `Europe/Tallinn` The mysql.timezone setting in config/database.php. If you do not specifically set it, then the database will probably use the system time of the server that it's running on.

1. We generate a date in our Laravel app using the `now()` helper function which returns us the following date: "2023-10-13 16:00:00". This is the current datetime in

`Europe/Berlin`

2. We then send “2023-10-13 16:00:00” to our MySQL database for storage in a timestamp column (for example by creating a Model and calling save() on it)

3. Our database takes “2023-10-13 16:00:00” and converts it to a Unix timestamp according to `Europe/Tallinn` timezone and then stores it. Notice what’s happening here? We generated the datetime according to `Europe/Berlin` but our database converted it to a timestamp according to `Europe/Tallinn`

4. When we retrieve the timestamp, our database converts the timestamp back to datetime according to `Europe/Tallinn` (session time_zone). Which results in “2023-10-13 16:00:00” (the original datetime we generated) So, at a glance everything seems to be ok. However, what happens if we change our app’s timezone to also be `Europe/Tallinn` ?

5. On retrieval, nothing changes, we still get back 2023-10-13 16:00:00 as the conversion depends on the database session time_zone and not on our app timezone.

6. Real issues arise when we start doing date comparisons in our app. Let’s say the date we originally saved was the creation date of a token and 30 minutes have passed since we generated it. We now wish to see whether the token is expired. For that:

- we get our current time with `now()` (which now generates dates according to `Europe/Tallinn` timezone as we changed our app’s timezone), we get 2023-10-13 17:30:00
- we get the token’s creation time from the database: 2023-10-13 16:00:00
- token should be valid for 1h, so we subtract the creation date from the current time and get a difference of 1.5h which seems to indicate that the token has expired. However in reality, only 30 minutes have passed.

## Key Takeaways and Best Practices

It might seem that running the database and the Laravel app in different timezones is pretty safe if you never change the timezone configs. However, this is a risky bet to make.

Timezone changes might easily happen if you are running many instances of your apps and databases. The majority of cloud providers set their instance timezones to UTC by default so if you are running a different timezone you need to be extra careful to always set the instances to that sepcific timezone.

You'll also need to consider daylight saving times. For example if your database session time_zone is UTC and your app timezone is `Europe/Berlin` then you'll end up with a plethora of issues on the last Sunday of October when the offset of `Europe/Berlin` changes due to daylight saving time change.

All this considered, the sanest way to handle dates in Laravel and MySQL is as follows:

Always set the timezones of your apps and databases to UTC. This way you'll not have to deal with any conversion and timezone issues.

If you wish to display dates according to your end-user's timezone then convert the date to end-user's timezone just before displaying it. Avoid storing it in a different timezone.

As for whether to use DATETIME or TIMESTAMP - that decision is up to you and is use case dependent.

eIDeasy

**EID Easy OÜ**
Telliskivi tn 60/1
Tallinn, 10412
Estonia
14080014
VAT: EE102310148
EE367700771002083464

PRODUCT

eSignatures
eID Authentication
eSealing

Pricing
Supported Methods
Book a call
Request demo

DEVELOPERS

Documentation
Supported Methods
Guide
Full API reference
Status

RESOURCES

Blog
Case Studies
FAQ

COMPANY

About us
News
Contact us

Cookie Preferences Responsible Disclosure Privacy Policy Terms and Conditions