November 22, 2022 · 11 mins, 2077 words

Dealing With Money in Laravel/PHP: Best Practices

When working with money in your Laravel projects, whether it's product prices or invoice total order amounts, we need to be extremely careful not to miscalculate something. Luckily, there are best practices and tools to help us with that, let's explore them in this article.

What we'll cover:

- Typical (and wrong) Money Behavior: Floats
- One Step Better: Integers
- Special PHP Packages For Money
- Laravel Way: Custom Casts
- Currency Conversion
- Specific Laravel Packages/Wrappers

Let's get into it!

Typical (and wrong) Money Behavior: Floats

How do you save data in the database when you need to work with things like product price, or order total?

Since the amount of money is a float – like **$3.45**, the logical way would be to store it the same way in the database.

```
$table->decimal('price', 8, 2);
// 8 is total digits, 2 is decimal digits
```

And then, whenever you need to show the price in Blade, you do something like:

```
Total price: ${{ number_format($product->price, 2) }}
```

In the code above, we need ``` `number_format()` ``` so that 9.1 would be shown as 9.10, with two digits.

In most cases, this approach should work fine, as long as you have one currency in your project and you're not doing a lot of **calculations** with those fields.

But the way how programming languages and database engines work with calculating floats, you may encounter a rounding problem. Meaning, you may have incorrect calculations by 0.01 if the wrong roundings add up.

Here are a few articles to read more about this problem:

- Floating Point Rounding Errors in MySQL
- Why You Should Never Use Float and Double for Monetary Calculations
- Floating Point Numbers & Currency Rounding Errors

If you are not in the mood to read those articles, and if you want to just trust me, I will simplify it for you:

NEVER STORE MONEY VALUES AS FLOATS IN THE DATABASE.

There, I said it.

Here's another quote for you:

Bill Karwin
@billkarwin

If I had a dime for every time I've seen someone use FLOAT to store currency, I'd have $999.997634. #ieee754jokes

6:49 AM · Jun 20, 2013 · Twitter Web Client

Yes, the possibility of that rounding error happening is very low, but still, it's better to be on the safe side, right? So here are the solutions below.

One Step Better: Integers

Instead of saving money as floats like dollars with cents, you should store them as cents only.

So, instead of having a float/decimal field with a value like `1.23`, you create an integer field with the value `123`.

At first, it may feel weird. In real life no one is calculating money in cents, right? But don't worry, we won't show it on the page as cents. We will transform them while getting from the database and set the cents amount before saving them into the database.

In Laravel, for this, you would typically use Model Attributes: Accessors and Mutators.

```
class Product extends Model
{
    protected function price(): Attribute
    {
        return Attribute::make(
            get: fn ($value) => $value / 100,
            set: fn ($value) => $value * 100,
        );
    }
}
```

Or, in the older syntax (*which still works in the latest Laravel version*):

```
class Product extends Model
{
    protected function getPriceAttribute($value)
    {
        return $value / 100;
    }

    protected function setPriceAttribute($value)
    {
        $this->attributes['price'] = $value * 100;
    }
}
```

So, when someone fills out the form and enters `` `123.45` `` as a value, it is transformed into `` `12345` `` in the database. And when the value needs to be shown later in some table, it is transformed back from a database integer value to a human-friendly `` `123.45` ``.

Now, there are a few exceptions to notice.

Exception no.1: not all the world currencies have **two** digits, like cents or pennies. According to this Wikipedia article, there are at least 9 world countries with currencies that have 3-4 decimal digits: Tunisian dinar, Bahraini dinar, and others. If you work with those currencies, then you still need to save data in integer, just divide/multiply by 1000 in case of 3 digits, and by 10,000 in case of 4 digits.

Exception no.2: likewise, some countries have NO decimal digits at all, the same Wikipedia article lists 17 countries like this. For those, no transformations are needed at all, just save money amount as it is.

Exception no.3: you may want to store more decimal numbers if it's needed according to your accounting logic. For example, the price of some very price-sensitive items may be not `` `$0.01` ``, but `` `$0.0123` ``, and then rounding the total price at the very last step of the purchase. Then you would store `` `123` `` in the database, so you work with the maximum needed numbers of decimal digits, in the lowest denomination.

But even with those exceptions, you get the idea: save the value as an integer, and multiply/divide every time. Works quite well, until it may be not enough.

Special PHP Packages For Money

So far, we've been looking at money as just a **number**: integer or float. But in real life, money is a more complicated object: what about **currency** and its rate?

Sure, your project may deal with only one currency, but what about multiple currencies? How to save data then: two fields in the DB? How to perform calculations?

A solution to all those questions lies in a term called **value object**, or a similar term is **data transfer object**. Those require a separate tutorial (*planned it in the future*), but, in short, it means creating an object with properties inside. Money is a very suitable example of a value object.

A quick example from MoneyPHP package:

```php
use Money\Currency;
use Money\Money;

$fiver = new Money(500, new Currency('USD'));
```

So, we create a `` `Money` `` object, which allows us later to transform that object into whatever we want, with many features of the PHP package.

```php
$value1 = Money::EUR(800);               // €8.00
$value2 = Money::EUR(500);               // €5.00
$value3 = Money::EUR(600);               // €6.00

$result = $value1->add($value2, $value3); // €19.00
```

There are two popular packages to deal with money in PHP

- moneyphp/money
- brick/money

Here's the example of that second package `brick/money`:

```
use Brick\Money\Money;

$money = Money::of(50, 'USD');

echo $money->plus('4.99'); // USD 54.99
echo $money->minus(1); // USD 49.00
echo $money->multipliedBy('1.999'); // USD 99.95
echo $money->dividedBy(4); // USD 12.50
```

A more interesting example:

```
$money = Money::of(100, 'USD');
[$a, $b, $c] = $money->split(3); // USD 33.34, USD 33.33, USD 33.33
```

On the surface, both packages perform the same thing: transforming the initial money value into an object, with many transformation features available to use with that object.

There's an interesting short comparison between two packages, in this Github comment. So I will stick to that comment and will show the `brick/money` package from here. How would it look in a typical Laravel project?

As you may have understood, storing data in the database doesn't change: you still store it in an integer. And if you work with multiple currencies, you store the currency code, too.

So, in the database migrations we have this:

```
Schema::create('orders', function (Blueprint $table) {
    $table->id();
    $table->integer('price');
    $table->string('currency')->default('USD');
    $table->timestamps();
});
```

Then, if we have an order with price `7907` (*meaning 79 dollars and 7 cents*) and currency `USD` in the database, we can have this Controller:

```
public function show(Order $order)
{
    return view('orders.show', [
        'id' => $order->id,
        'price' => Money::ofMinor($order->price, $order->currency)->formatTo('en_US
    ]);
}
```

In the blade, we show the data like this:

```
Order ID: {{ $id }} ({{ $price }})
```

Result: "Order ID: 1 ($79.07)"

As you can see, we didn't put the $ sign upfront, we didn't divide by 100, and we didn't format anything manually. The Money package takes care of everything, we just need to call `` `->formatTo('en_US')` `` with the locale we want.

Convenient, isn't it?

Laravel Way: Custom Casts

On top of those PHP packages, Laravel gives us even more power: we don't need to create `` `Money::ofMinor()` `` every time, and we can use Custom Casts, so that `` `$order->price` `` field would automatically be transformed to a Money object.

We run:

```
php artisan make:cast Money
```

It generates the file **app/Casts/Money.php** which we fill like this:

```
class Money implements CastsAttributes
{
    public function get($model, string $key, $value, array $attributes)
    {
        return \Brick\Money\Money::ofMinor($attributes['price'], $attributes['curre
    }

    public function set($model, string $key, $value, array $attributes)
    {
        if (! $value instanceof \Brick\Money\Money) {
            return $value;
        }

        return $value->getMinorAmount()->toInt();
    }
}
```

And then we assign that class to the Model.

app/Models/Order.php:

```
use App\Casts\Money;

class Order extends Model
{
    protected $casts = [
        'price' => Money::class
    ];
}
```

Finally then, in the Controller, we don't need to perform any transformations, we can just pass the `` `$order` `` object, and perform formatting in the Blade:

```
class OrderController extends Controller
{
    public function show(Order $order)
    {
        return view('orders.show', compact('order'));
    }
}
```

Blade file:

```
Order ID: {{ $order->id }}
<br />
Price: {{ $order->price->formatTo('en_US') }}
```

As you can see, `` `$order->price` `` is already a Money object, and we can use `` `->formatTo()` `` directly on that.

Similarly to how Laravel by default has `` `created_at` `` and `` `updated_at` `` fields as Carbon objects, so with timestamps in a Blade file, we can do something like `` `{{ $order->created_at->diffForHumans() }}` ``

Notice: in the Cast class, there's a `` `get()` `` and a `` `set()` `` method. The first one is clear, but the second one `` `set()` `` is more tricky, because it depends on what is passed to the price field: in some cases, it may be just an integer (then we just return it as `` `$value` ``), but maybe you would have it as Money object, then you would need to perform transformation like `` `$value->getMinorAmount()->toInt()` ``.

Currency Conversion

Now, we're storing the currency, but what's the use of it if we don't convert it to other currencies?

Of course, the topic of conversion is huge in itself and worth a separate long article, but let's see what possibilities we have in the `` `brick/money` `` package, for example.

The package is shipped with a specific class `` `Brick\Money\CurrencyConverter` ``, which accepts an exchange rate provider parameter. Several implementations of the rate provider are in the package.

ConfigurableProvider: This provider starts with a blank state, and allows you to add exchange rates manually.

So, in the model, we define this Accessor:

```
use Brick\Math\RoundingMode;
use Brick\Money\CurrencyConverter;
use Brick\Money\ExchangeRateProvider\ConfigurableProvider;

class Order extends Model
{
    public function getPriceEurAttribute()
    {
        $exchangeRateProvider = new ConfigurableProvider();
        $exchangeRateProvider->setExchangeRate('USD', 'EUR', '0.9123');
        $converter = new CurrencyConverter($exchangeRateProvider);

        return $converter->convert(
            moneyContainer: $this->price,
            currency: 'EUR',
            roundingMode: RoundingMode::DOWN
        );
    }
}
```

This Accessor returns `` `$order->price_eur` `` as a Money object, and then in the Blade, we can do something like this:

```
Price: {{ $order->price->formatTo('en_US') }}
({{ $order->price_eur->formatTo('en_US') }})
```

Result: "Price: $57.15 (€52.13)"

Also, there are other Currency Providers:

- **PDOProvider**: reads exchange rates from a database table
- **BaseCurrencyProvider**: for the quite common case where all your available exchange rates are relative to a single currency

Also, you can write your own provider, implementing the `` `ExchangeRateProvider` `` interface and the method `` `getExchangeRate()` ``.

With all those providers, you can also implement to get the exchange rates from external APIs like this one, or datasets like this XML provided by the European Central Bank, and crawl the data regularly from there into your database.

Specific Laravel Packages/Wrappers

Finally, we get down specifically to Laravel tools around money. So far, we've been discussing `` `brick/money` `` and `` `moneyphp/money` `` which are PHP packages, is there anything for Laravel?

Glad you asked.

akaunting/laravel-money – This package intends to provide tools for formatting and conversion of monetary values in an easy, yet powerful way for Laravel projects.

It's a standalone package, meaning it's not a wrapper over any PHP package from above. On top of creating the Money object, it also provides Helpers, Blade directives, and Components:

```
Money::USD(500);
money(500, 'USD')
@money(500, 'USD')
<x-money amount="500" currency="USD" />
<x-currency currency="USD" />
```

Also, there are a few wrapper packages like cknow/laravel-money which abstracts MoneyPHP mentioned above, also adding Laravel features like custom casts, helpers, and Blade directives.

Conclusion: Money is Hard

These are just a few possible solutions and tools how to store and format money value in your Laravel projects.

Real-life scenarios are even more difficult: we haven't covered examples like calculating the invoice values with taxes and fraction values of the items, rounding in specific conditions, and many more parameters provided by the packages above.

But I hope this article will give you enough overview to understand the ecosystem and the context, and then you would dive deeper based on your specific project needs.

Get notified when participating

David Carr
5 months ago

great article, I've being sting in the past storing money as a string or floats then wondering why calulations are out with rounding.

Leave a reply

Haytham Abdulla
5 months ago

Thank Mr. Povilas. Great article.

I did the casting, the display is working fine, I used this:

```
use Brick\Money\Money as BrickMoney;
function priceShow(): Attribute
    {
        return Attribute::make(
            get: function () {
                $price = BrickMoney::ofMinor($this->product_price, $this->p
                return $price;
            }
        );
    }
```

So I'm getting the price in my blade directly.

But for saving the price, I didn't figure out how to do it, I did the casting and my table price column is integer. For example when I save 27.8, it is saved in the database as 28.

Haytham Abdulla
5 months ago

Got it:

```
// to get the price formated
function priceShow(): Attribute
    {
        return Attribute::make(
            get: function () {
                return BrickMoney::of($this->product_price, $this->produ
            }
        );
    }

// to set and get the price
    protected function productPrice(): Attribute
    {
        return Attribute::make(
            get: fn ($value) => BrickMoney::ofMinor($value, $this->produ
            set: fn ($value) => BrickMoney::of($value, $this->product_cu
        );
    }
```

Haytham Abdulla

5 months ago

sorry for keeping posting. I'm using Nova, I'm trying to make this work with my frontend and my backend which is Nova admin panel.

Now I'm using: in the Order modle:

```
function priceShow(): Attribute
    {
        return Attribute::make(
            get: function () {
                return BrickMoney::ofMinor($this->product_price, $this->
            }
        );
    }


    function priceClean(): Attribute
    {
        return Attribute::make(
            get: function () {
                return BrickMoney::ofMinor($this->product_price, $this->
                ->getAmount()->toFloat();
            }
        );
    }
```

in the observer:

```
public function saving(Order $order)
    {
        $order->product_price = BrickMoney::of($order->product_price, $c
    }
```

Povilas Korop

5 months ago

Sorry I'm not a Nova user so I'm not sure what that tool does with the fields automatically, that may affect the code and may be different from my example.

I think you should look at Nova docs and how it deals with Money, and follow my article only for theoretical knowledge how it works under the hood.

Haytham Abdulla

5 months ago

Thanks! For any one looking to display the curreny in the right format, the currency field withen Nova have functions to do that:

```
Currency::make('Estimated Reward')
                    ->hideFromIndex()
                    ->rules('required', 'numeric')
                    ->currency('BHD')
                    ->asMinorUnits()
```

you can do somthing like this also:

```
->displayUsing(function ($value) {
                        return BrickMoney::ofMinor($this->product_price,
                    })
                    ->resolveUsing(function ($value) {
                        return BrickMoney::ofMinor($value, $this->produc
                    })
```

Leave a reply

Akaunting
4 months ago

I'd be more relaxed with saving floats in the database. It depends. For example, you don't show arbitrary precision amounts in the accounting industry but always round them. Of course, you don't do rounding on the database side but in PHP.

And in the case of Akaunting, because users can change the precision from UI whenever they want, you can't save it as an integer. That's why we made the `akaunting/laravel-money` package, which works like a charm for 200K+ Akaunting users ;)

Best regards, Denis Duliçi

Leave a reply

Huthaifah Kholi
4 months ago

Great,

what about DB migration , what is the column type of price at database ?

Povilas Korop
4 months ago

Integer.

Leave a reply

Rodrigo Borges
3 months ago

Very cool this article. How do you suggest using integrated with livewire? For example, how would the livewire property look to show the formatted value in the input and then change it to an integer to persist in the base?

Povilas Korop
3 months ago

Interesting question. I would probably do the conversion on Eloquent level, with Casts and accessors/mutators if needed, Livewire would be the layer just for **presenting** the data.

Leave a reply

Huy Nguyen
1 month ago

Great article! Looking forward to more articles that dives deep more about how to use these packages for complex calculations, currency conversion based on various business requirement scenarios

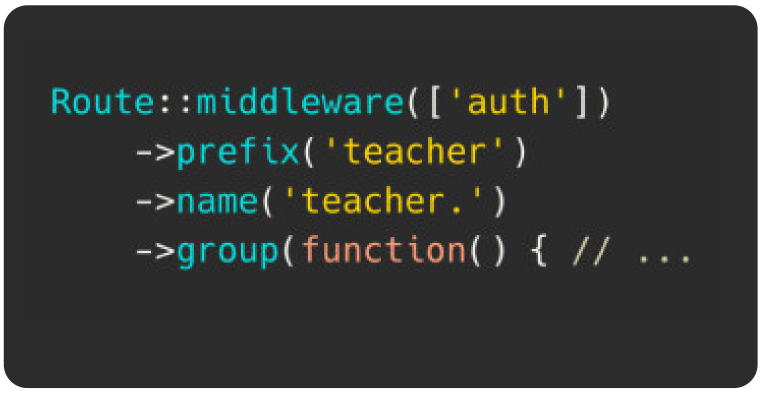
Leave a reply

Leave a comment

You can use Markdown

Comment

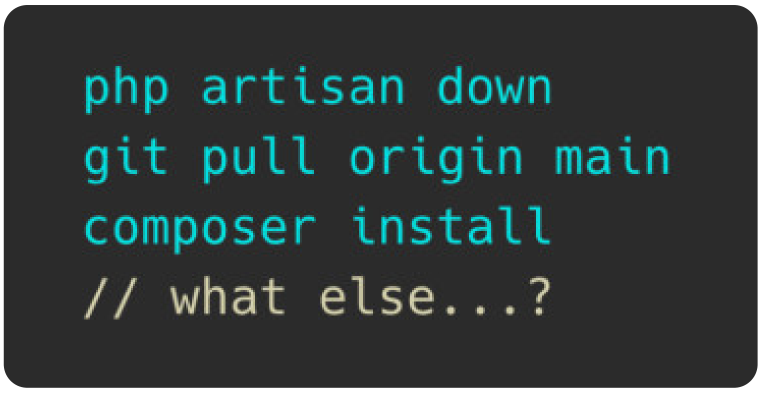
Recent Premium Tutorials



```
Route::middleware(['auth'])
    ->prefix('teacher')
    ->name('teacher.')
    ->group(function() { // ...
```

October 20, 2022 · 25 mins, 4982 words · ⭐ PREMIUM

Laravel Breeze with User Areas: Student, Teacher, Admin



```
php artisan down
git pull origin main
composer install
// what else...?
```

November 24, 2022 · 6 mins, 1121 words · ⭐ PREMIUM

Laravel Deployment Script: 4 Steps to Add Changes on Server

```
class Post extends Model {
    use HasUuid;
}
```

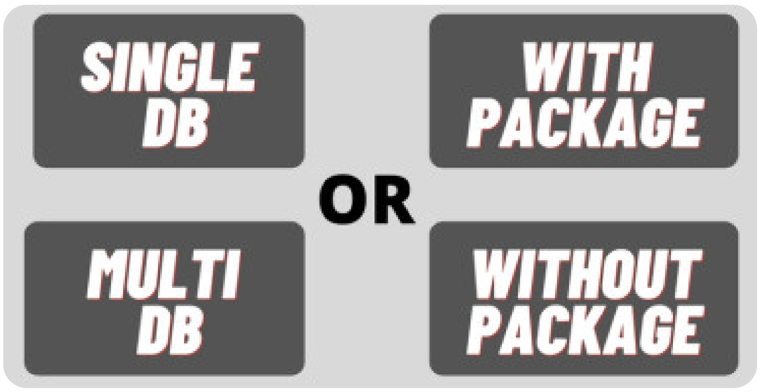
December 08, 2022 · 9 mins, 1656 words · ⭐ PREMIUM

UUID in Laravel: All You Need To Know

```
$url = "https://api.ipdata.co/...";
$result = Http::get($url)->json();
```

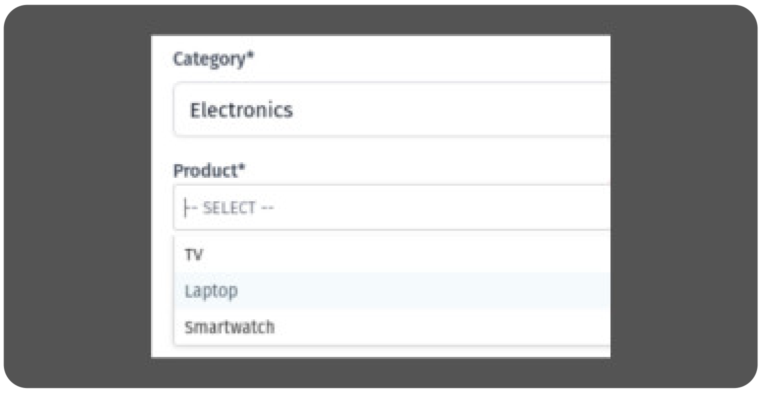
January 05, 2023 · 11 mins, 2004 words · ⭐ PREMIUM

Laravel Testing: Mocking/Faking External 3rd Party APIs



February 07, 2023 · 9 mins, 1748 words · ⭐ PREMIUM

Laravel Multi-Tenancy with Multi-Database: Step-by-Step Example



April 18, 2023 · 17 mins, 3321 words · ⭐ PREMIUM

Livewire Parent-Child Dropdowns: 2-Level, 3-Level, and Select2 Alternatives