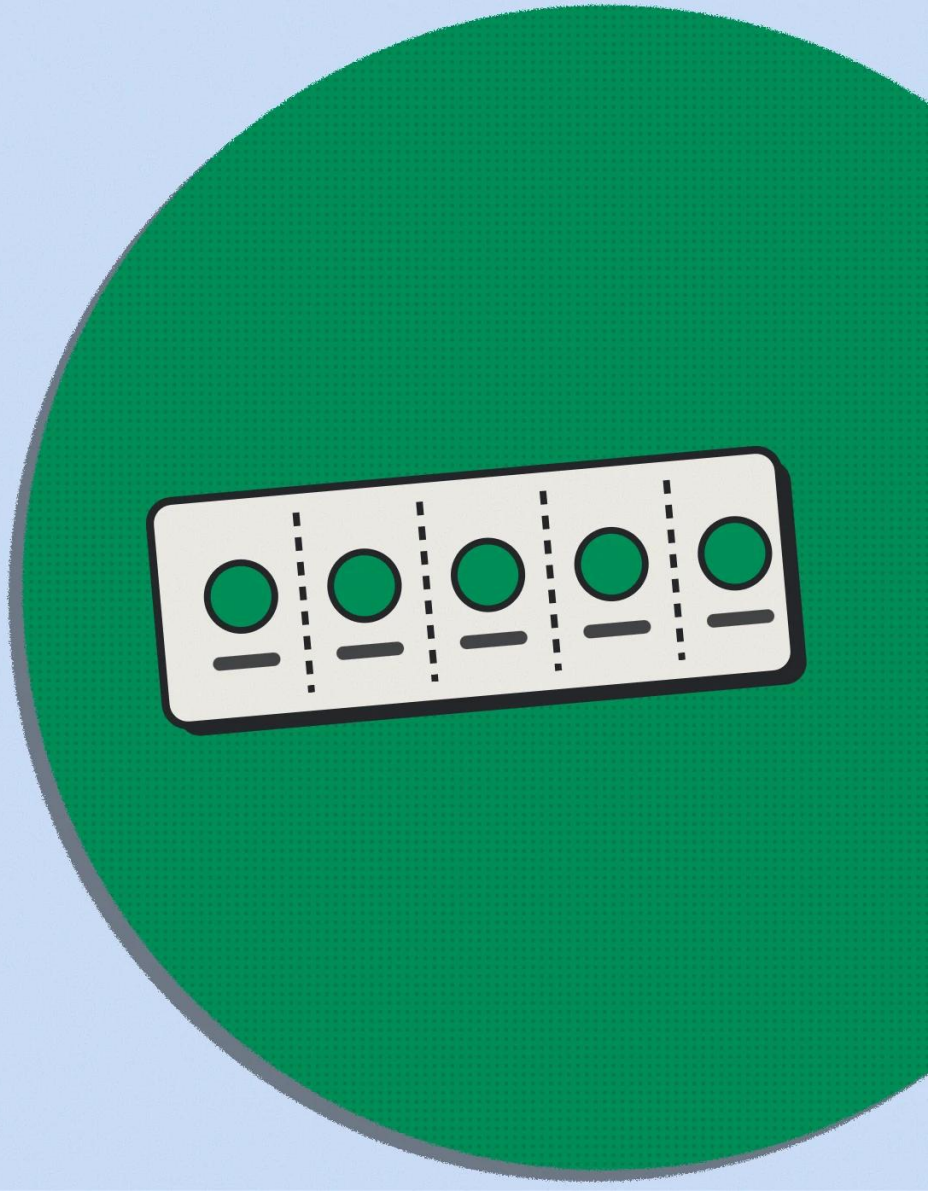
2.07

TEMPS 2 · LE TAMAGOTCHI TROUÉ · SÉANCE 2

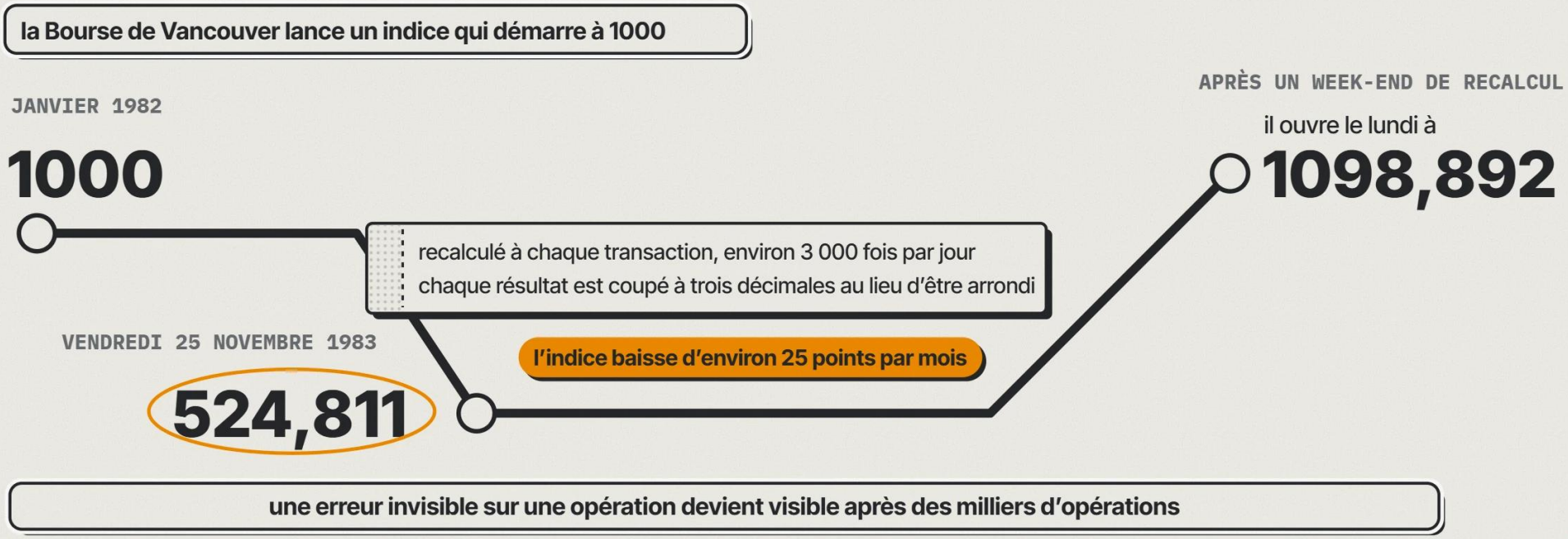
Les centimes



Thibault Six · IFOSUP Wavre · 2026-2027

? Une erreur de moins d'un millième par calcul : peut-elle vraiment coûter quelque chose ?

À Vancouver, une coupure répétée à chaque transaction a fait tomber l'indice de 1000 à 524,811.



Chaque coupure perdait moins d'un millième de point. Répétée des milliers de fois, elle a fait chuter l'indice, sans que les cours des actions baissent.

? Que répond PHP à `0.1 + 0.2` ? Et `0.1 + 0.2` est-il égal à `0.3` ?

`echo` affiche `0.3`, mais `var_dump()` montre que, pour PHP, `0.1 + 0.2` ne vaut pas `0.3`.

un nombre à virgule s'appelle en PHP un float

ECHO

```
$ php artisan tinker --execute="echo 0.1 + 0.2;"
0.3
```

`echo` affiche au plus quatorze chiffres significatifs et cache l'écart

VAR_DUMP()

```
$ php artisan tinker --execute="var_dump(0.1 + 0.2, 0.1 * 3, 0.1 + 0.2 == 0.3);"
float(0.30000000000000004)
float(0.30000000000000004) ── 0.1 * 3
bool(false) ── 0.1 + 0.2 == 0.3 rend false
```

`var_dump()` en affiche assez pour le montrer

Même calcul, deux affichages. `echo` arrondit l'écriture et tout paraît juste ; `var_dump()` montre le nombre que PHP a vraiment calculé.

? Pourquoi PHP ne sait-il pas ranger 0,1 exactement ?

En binaire, 0,1 s'écrit sans fin, comme un tiers en décimal : PHP en garde une valeur arrondie.

EN DÉCIMAL : UN TIERS

0,3333… vous devez vous arrêter quelque part

EN BINAIRE : UN DIXIÈME

0,00011001100… le motif `0011` se répète sans fin

PHP

PHP range un float sur 64 bits, en binaire, avec un nombre fixe de chiffres

PHP arrondit 0,1 au nombre binaire le plus proche qu'un float sait écrire

```
$ php artisan tinker --execute="printf('%.20f', 0.1);"
0.10000000000000000555
```

le 0.1 de PHP vaut un tout petit peu plus que 0,1

0.2 et 0.3 ont aussi leur petit écart, et ces écarts ne s'annulent pas toujours dans un calcul

En haut, ce que vous connaissez déjà ; en bas, le même problème en binaire. PHP garde le nombre le plus proche, pas 0,1.

? Bob a 0,30 € et achète trois bonbons à 0,10 €. Que répond le magasin ?

Le magasin refuse Bob : pour PHP, trois fois `0.1` dépasse de très peu son solde de `0.3`.



ONGLET RÉSEAU

```
POST /magasin/1/acheter
{"price":"0.1","creature_id":"5","quantity":"3"}
Réponse : 302, retour à /magasin
```

Le chapitre 2.08 s'occupe du prix qui vient de la requête.

app/Http/Controllers/PurchaseController.php — extrait

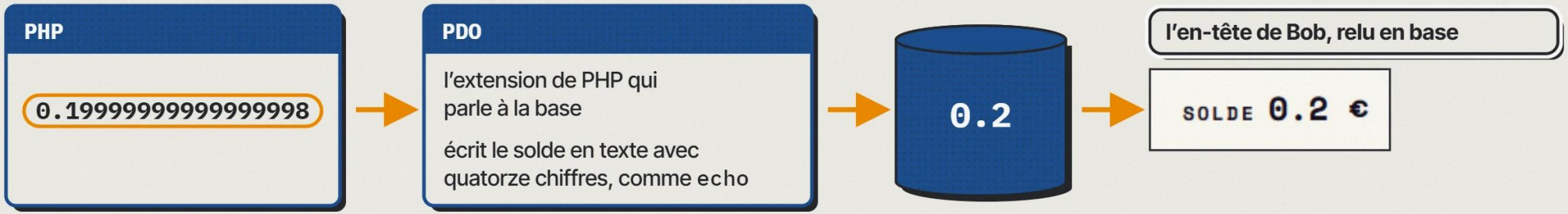
```php
$quantity = $request->integer('quantity', 1);
$price = $request->float('price');
$total = $price * $quantity;    0.1 * 3, soit 0.30000000000000004
// …
if ($user->balance < $total) {    0.3 < 0.30000000000000004 est vrai
    return back()->withErrors(['quantity' => 'Votre solde est insuffisant.']);
}
```

À gauche, ce que Bob voit ; à droite, ce que PHP calcule. Le total dépasse le solde d'un écart que l'écran ne montre pas.

? Si l'écart existe, pourquoi le développeur ne l'a-t-il jamais vu, ni à l'écran ni en base ?

PHP calcule un solde à peine sous `0.2`, mais PDO écrit `0.2` en base : l'écart reste caché.

APRÈS UN ACHAT D'UN BONBON



À VOUS : REFAITES LE CALCUL 5 min

1. Achetez trois bonbons d'un coup, et relevez le message sous la quantité.
2. Achetez-les un par un, et regardez le solde de l'en-tête.
3. Lancez `php artisan test --filter=Flaw07` et lisez les noms des quatre tests rouges.

Suivez le solde : l'écart disparaît chez PDO. On ne le voit que si le total égale le solde.

? Comment calculer trois fois 0,10 € sans le moindre écart ?

En centimes, `10 * 3` vaut exactement `30` : la base range des entiers, le serveur écrit les euros.

	AVANT	APRÈS
EN BASE	`balance = 0.3` un float	`balance_cents = 30` un entier
LE CALCUL, EN PHP	`0.1 * 3 = 0.30000000000000004` l'achat est refusé	`10 * 3 = 30` l'achat passe
LA PAGE, AVANT L'ACHAT	`resources/js/components/AppHeader.vue` `{{ $page.props.auth.user.balance }} €` → `0.3 €` l'en-tête colle le signe euro derrière le nombre reçu	le serveur → `0,30 €` le serveur envoie à la page une chaîne déjà écrite

le suffixe `_cents` dit l'unité dans le nom de la colonne

Chaque rangée compare l'avant et l'après. À droite, aucun nombre à virgule avant l'écran.

? Pour passer 0,29 € en centimes, suffit-il de multiplier par 100 et de garder l'entier ?

La migration arrondit avec `ROUND`, car convertir en entier couperait 0,29 € à 28 centimes.

```
$ php artisan tinker --execute="var_dump(0.29 * 100, (int) (0.29 * 100), round(0.29 * 100));"
float(28.999999999999996)
int(28)
float(29)
```

`int(28)` — **convertir en entier coupe au lieu d'arrondir : 0,29 € deviendrait 28 centimes**

database/migrations/…_convert_amounts_to_cents.php — extrait

```
DB::table('users')->update(['balance_cents' => DB::raw('ROUND(balance * 100)')]);
```

`DB::raw()` passe une expression SQL telle quelle, et la base calcule elle-même, ligne par ligne

SQLite se comporte de même : `CAST(0.29 * 100 AS INTEGER)` y vaut 28, `ROUND(0.29 * 100)` vaut 29

LES MONTANTS DANS VOTRE BASE

AVANT

```
{"Alice":120,"Mallory":5,"Bob":0.3}
```

php artisan migrate →

APRÈS

```
{"Alice":12000,"Mallory":500,"Bob":30}
```

En haut, la conversion qui coupe ; au milieu, la ligne qui arrondit en SQL ; en bas, les soldes migrés.

? Le test de Money::format() attend '10,50 €' et reçoit '10,50 €'. Pourquoi est-il rouge ?

`Number::currency()` met une espace insécable avant le € : le test doit écrire ce caractère-là.

1 Il reste l'affichage, le second défaut vu à la section 3 : la page écrit `0.3 €` au lieu de `0,30 €`. `Money::format()` reçoit des centimes et rend une chaîne : `1050` donne `10,50 €`. Laravel fournit `Number::currency()`.

Vous écrivez donc d'abord son test.

2 `php artisan test tests/Unit/MoneyTest.php`

```
--- Expected
+++ Actual
@@ @@
-'10,50 €'
+'10,50 €'
```

À l'écran, les deux s'écrivent `10,50 €`. Pourtant `toBe()` les dit différentes :

un caractère diffère, et il ne se voit pas

3



U+202F espace fine insécable, entre les milliers

U+00A0

U+00A0, espace insécable : un caractère qui ressemble à une espace, mais qui interdit de couper la ligne à cet endroit. Le montant et son signe € restent ainsi toujours sur la même ligne.

Votre clavier tape une espace ordinaire, U+0020.

4 **Ne touchez pas à `Money::format()` : c'est la valeur attendue du test qui doit changer.**

tests/Unit/MoneyTest.php

```
[123456, "1\u{202F}234,56\u{00A0}€"],
```

Entre guillemets simples, PHP laisse les huit caractères tels quels.

À l'écran, rien ne change.

À l'écran, rien ne change. Dans la valeur attendue du test, écrivez `\u{00A0}` avant le signe € et `\u{202F}` entre les milliers.

? Qu'est-ce qui doit être vrai à la fin ?

À vous : rangez et calculez les montants en centimes, jusqu'aux quatre tests verts de `Flaw07`.

À FAIRE

1. la migration
2. les modèles, les fabriques et le seeder
3. les contrôleurs
4. un test unitaire pour `Money::format()`
5. le serveur envoie des montants écrits

À VÉRIFIER

```
$ php artisan test --filter=Flaw07
  Tests:    4 passed (20 assertions)
```

```
$ php artisan test
  Tests:    8 failed, 81 passed (347 assertions)
```

SOLDE 0,00 €

Achat effectué pour Noisette.

PRIX 0,10 €

STOCK
197 en stock

Cherchez dix minutes, puis ouvrez la correction.

Rejouez l'achat de Bob : trois bonbons d'un coup passent.

DANS VOTRE CARNET DES FAILLES

Un montant se range et se calcule en centimes entiers, et le serveur ne le met en forme qu'au moment de l'afficher.

Cinq étapes à gauche ; à droite, les tests, puis l'achat de Bob qui passe. La règle va dans votre carnet, mot pour mot.