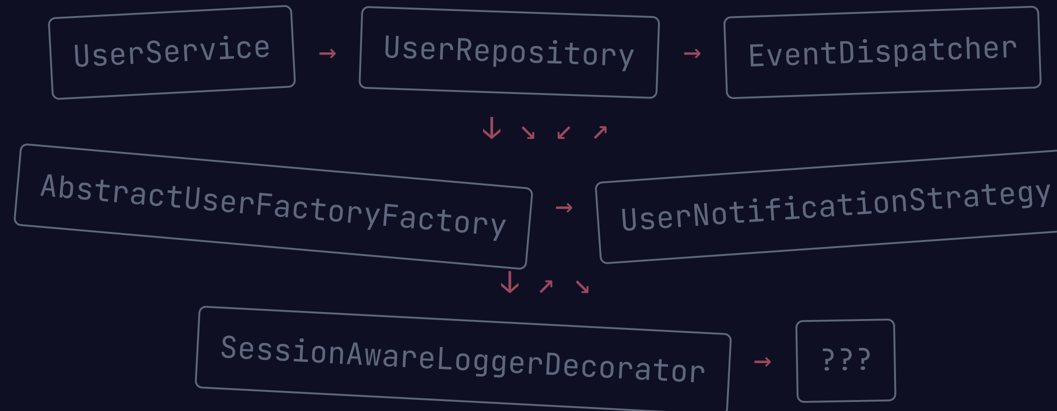
// MODERN PHP

Beyond ~~OOP~~ Functional Patterns in Modern PHP

Pure functions. Immutability. Composition. Result types. Without dogma.

php[tek] 2026 • 50 minutes • PHP 8.5

We've all been here.



Six classes deep, four interfaces, and the bug is in `__construct`.

”You wanted a banana,
but you got a gorilla
holding the banana
and the entire jungle.”

— Joe Armstrong, on OOP inheritance

// WHAT THIS TALK IS

PHP is multi-paradigm.

We're not throwing out OOP. We're stealing the parts of FP that make your code easier to test, easier to reason about, and easier to change.

`pure functions` `immutability` `composition` `map/filter/reduce` `pipelines` `result types`

// CHAPTER 1

01/

The Modern PHP Functional Toolkit

Your language already supports this. PHP 7.4 → 8.5.

// PHP 8.1+

First-Class Callable Syntax

```php
// before 8.1
$upper = fn($x) => strtoupper($x);

// 8.1+
$upper = strtoupper(...);

$names = ['alice', 'bob'];
$loud  = array_map(strtoupper(...), $names);
```

Functions are values. Pass them, store them, compose them.

// PHP 7.4+

Arrow Functions

```php
// verbose
$double = function ($n) use ($factor) {
    return $n * $factor;
};

// arrow — auto-captures by value
$double = fn ($n) => $n * $factor;
```

Single expression. No `use`. Perfect for `map/filter/reduce`.

// PHP 8.0+

Match — an Expression, not a Statement

```php
$label = match (true) {
    $age < 13 => 'child',
    $age < 20 => 'teen',
    $age < 65 => 'adult',
    default   => 'senior',
};
```

- Returns a value — assign it, return it, pipe it
- Strict comparison (`===`) by default
- No fall-through, no `break`

// PHP 8.1 PROPS / 8.2 CLASSES

Readonly — Immutability Built In

```
readonly class Money {
    public function __construct(
        public int    $amount,
        public string $currency,
    ) {}
}

$price = new Money(1000, 'USD');
$price->amount = 2000;  // Fatal error!
```

Once constructed → locked.

// PHP 8.5 — NOVEMBER 2025

The Native Pipe Operator

```php
// inside-out, hard to read
$slug = trim(strtolower('  Hello World  '));

// left-to-right with ▷
$slug = '  Hello World  '
      ▷ strtolower(...)
      ▷ trim(...);
```

`$x ▷ $f` is just `$f($x)` — but composition becomes obvious.

// CHAPTER 2

02/

Pure Functions — The Foundation

The single most impactful idea you can steal from FP.

data in,
data out.

No mutations. No logger. No events. No `$this`.

Pure Function — Two Rules

1

Same input → same output

Always. Forever. Deterministic.

2

No side effects

No DB, no I/O, no globals, no clock.

Spooky vs Pure

// IMPURE

```php
class OrderService {
  public function calcTotal(array $items): float {
    $this->logger->info('calc');
    $tax = $this->config->tax();
    $sub = 0;
    foreach ($items as $i) {
      $sub += $i->price * $i->qty;
      $i->processed = true;
    }
    Event::fire('totaled');
    return $sub + ($sub * $tax);
  }
}
```

// PURE

```php
function calcOrder(
    array $items,
    float $tax
): array {
    $sub = array_sum(array_map(
        fn($i) => $i->price * $i->qty,
        $items
    ));
    $taxAmount = round($sub * $tax, 2);
    return [
        'subtotal' => $sub,
        'tax'      => $taxAmount,
        'total'    => $sub + $taxAmount,
    ];
}
```

// NO MOCKS. NO FIXTURES. NO SETUP.

Testing Nirvana

```
public function testCalculateOrderTotal(): void
{
    $items = [
        ['price' => 10.00, 'quantity' => 2],
        ['price' =>  5.00, 'quantity' => 3],
    ];
    $result = calculateOrderTotal($items, taxRate: 0.08);

    $this->assertEqualsWithDelta(35.00, $result['subtotal'], 0.001);
    $this->assertEqualsWithDelta( 2.80, $result['tax'],      0.001);
    $this->assertEqualsWithDelta(37.80, $result['total'],    0.001);
}
```

That's the whole test.

// CHAPTER 3

03/

Immutable Data — When It Makes Sense

Don't fight your ORM. Do lock down your value objects.

”Who changed my object?”

```
$user = $repo->find(42);
$this->auditLog->record($user);     // mutates email!
$this->notifier->send($user);       // mutates name!
$this->analytics->track($user);     // ¿qué?

// You: "why is the welcome email empty?"
```

Mutation at a distance. The hardest bug to find.

// READONLY + NAMED ARGUMENTS

Immutable DTOs

```php
readonly class CreateOrderRequest {
    public function __construct(
        public int    $customerId,
        public array  $items,
        public string $shippingAddress,
        public ?string $couponCode = null,
    ) {}
}

$req = new CreateOrderRequest(
    customerId:      42,
    items:           [...],
    shippingAddress: '123 Main St',
);
```

The Wither Pattern (PSR-7 style)

```php
readonly class Address {
    public function __construct(
        public string $street,
        public string $city,
        public string $state,
        public string $zip,
    ) {}

    public function withCity(string $city): self {
        return new self(
            $this->street, $city, $this->state, $this->zip
        );
    }
}
```

Returns a new instance. Original is untouched.

// PHP 8.5

clone() with property overrides

```php
readonly class Address {
    public function __construct(
        public string $street, public string $city,
        public string $state,  public string $zip,
    ) {}

    public function withCity(string $city): self {
        return clone($this, ['city' => $city]);
    }
}
```

One line instead of five. Works on `readonly` properties from inside the class.

Practical Rules — Immutability

// USE IT FOR

- Value Objects (Money, DateRange)
- DTOs / API request & response
- Configuration objects

// DON'T FIGHT IT FOR

- Eloquent / Doctrine entities
- Long-lived objects with lifecycle
- Anything an ORM is tracking

If it's a value, lock it. If it's an entity, let it live.

// CHAPTER 4

04/

Map / Filter / Reduce

Without looking pretentious.

Tell what, not how

// IMPERATIVE

```php
$totals = [];
foreach ($orders as $o) {
    if ($o->status === 'paid') {
        $totals[] = $o->total * 1.08;
    }
}
$grand = 0;
foreach ($totals as $t) {
    $grand += $t;
}
```

// DECLARATIVE

```php
$paid = array_filter(
    $orders,
    fn($o) => $o->status === 'paid'
);

$grand = array_sum(array_map(
    fn($o) => $o->total * 1.08,
    $paid
));
```

// THE SWISS ARMY KNIFE

Reduce — Shape Anything

```php
// group orders by status
$grouped = array_reduce(
    $orders,
    function (array $acc, Order $o): array {
        $acc[$o->status] ??= [];
        $acc[$o->status][] = $o;
        return $acc;
    },
    []
);
// ['paid' => [...], 'pending' => [...], ...]
```

Map and filter are special cases of reduce.

⚠ The Pretentious Trap

Don't replace a five-line `foreach` with a fifteen-line nested `reduce`.

Rules of thumb:

- **Loop** — when you need an early `break`, complex flow, or side effects
- **map/filter/reduce** — when each step is one expression
- **Reach for a library** (`illuminate/collections`) when chaining gets ugly

// CHAPTER 5

05/

Composition & Pipelines

Lego bricks, not monoliths.

// LEAGUE/PIPELINE

The Pipeline Pattern

```php
use League\Pipeline\Pipeline;

$slugify = (new Pipeline)
    ->pipe(fn($s) => trim($s))
    ->pipe(fn($s) => strtolower($s))
    ->pipe(fn($s) => preg_replace('/[^a-z0-9]+/', '-', $s))
    ->pipe(fn($s) => trim($s, '-'));

$slug = $slugify->process('  Hello World!  ');  // "hello-world"
```

Top-to-bottom. Each stage testable in isolation. Reusable across the codebase.

// PHP 8.5 — NO LIBRARY REQUIRED

Or Use the Native Pipe

```php
$slug = '  Hello World!  '
    |> trim(...)
    |> strtolower(...)
    |> (fn($s) => preg_replace('/[^a-z0-9]+/', '-', $s))
    |> (fn($s) => trim($s, '-'));
```

// NATIVE |>

One-shot transformations. Inline. No setup.

// LEAGUE/PIPELINE

Reusable objects, DI-friendly, class-per-stage.

A Stage as an Invokable Class

```
class ValidateRows {
    public function __construct(
        private readonly ValidatorFactory $validator
    ) {}

    private const RULES = [
        'email' => 'required|email',
        'name'  => 'required|string',
    ];

    public function __invoke(array $rows): array {
        return array_values(array_filter(
            $rows,
            fn($r) => $this->validator->passes($r, self::RULES)
        ));
    }
}
```

Constructor-injected dependencies. `__invoke` stays pure-ish. DI container friendly.

// REALISTIC DATA IMPORT

One Pipeline, Five Stages

```
$import = (new Pipeline)
    ->pipe(new ParseCsv())
    ->pipe(new NormalizeRows())
    ->pipe(new ValidateRows($validatorFactory))
    ->pipe(new DeduplicateBy('email'))
    ->pipe(new PersistToDatabase($repo));

$stats = $import->process($uploadedFile);
```

Read it. You know what happens. Each stage tested alone.

// CHAPTER 6

06/

Handling ”Nothing” Without Losing Your Mind

The billion-dollar mistake, and how to stop making it.

”I call it my billion-dollar mistake.
It was the invention of the null reference in 1965.”

— Sir Tony Hoare, inventor of null

Three Ways PHP Handles "Nothing"

1. **Return `null`** — caller has to remember to check. They won't.
2. **Throw an exception** — but is "user not found" really exceptional?
3. **Return a `Result` object** — make the caller deal with both paths.

The Result Object

```
readonly class Result {
    private function __construct(
        public bool    $ok,
        public mixed   $value = null,
        public ?string $error = null,
    ) {}

    public static function success(mixed $v): self {
        return new self(ok: true, value: $v);
    }
    public static function failure(string $e): self {
        return new self(ok: false, error: $e);
    }
}
```

For real codebases, split into `Success`/`Failure` classes — or use `phpoption/phpoption`.

Forcing the Caller to Decide

```php
$result = findUser(42);

if ($result->ok) {
    sendWelcomeEmail($result->value);
} else {
    logWarning($result->error);
}
```

You can't accidentally use `$result` as a User.

The type system says: handle both paths, or your code won't compile.

// ALTERNATIVE

The Null Object Pattern

```php
class GuestUser implements UserInterface {
    public function name(): string         { return 'Guest'; }
    public function permissions(): array   { return []; }
    public function isAuthenticated(): bool { return false; }
}

$user = $repo->findOrGuest($id);
echo $user->name();  // never null. always safe.
```

Replace "absent value" with a real object that has sensible defaults.

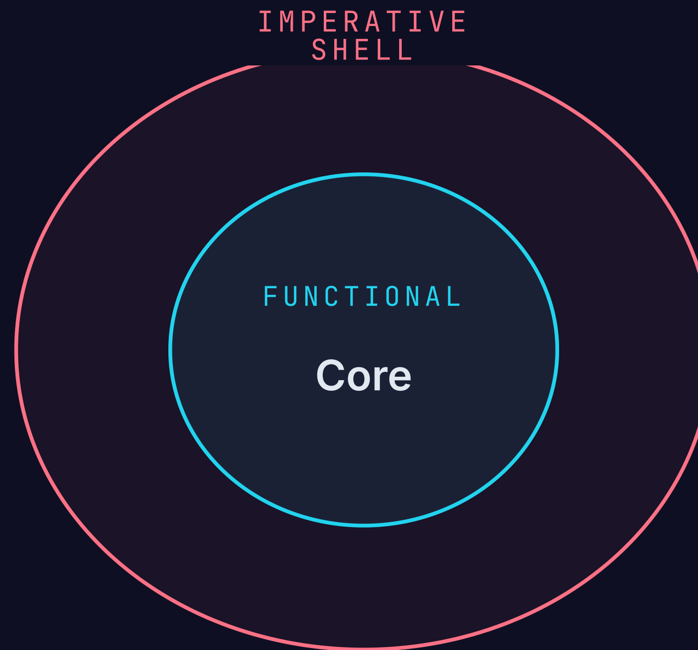
// CHAPTER 7

07/

Architecture: Imperative Shell, Functional Core

Where it all comes together.

Shell wraps Core



Shell — I/O, DB, HTTP, events. Where the world is messy.
Core — pure functions, immutable values. Easy to test, easy to reason about.

Shell + Core in Practice

```php
// SHELL — the controller (impure: reads, writes)
public function checkout(Request $request): Response {
    $cart   = $this->cartRepo->findFor($request->userId);
    $coupon = $this->couponRepo->find($request->couponCode);

    // CORE — pure calculation
    $totals = calculateOrderTotals($cart->items, $coupon, taxRate: 0.08);

    // SHELL again — persist, side-effect
    $this->orderRepo->persist($cart, $totals);
    return new Response($totals);
}
```

Shell is thin. Core is fat. **Tests live mostly in the core.**

Pragmatism over dogma.

You don't have to write Haskell. You don't have to throw out your framework. Steal the parts that pay rent.

// AFTER THIS TALK

Start Small. Pick One.

1. Find one calculation. Make it pure.
2. Lock one DTO with `readonly`.
3. Replace one nested loop with `map/filter/reduce`.
4. Wrap one nullable return in a `Result`.
5. Pipe one transformation with `▷`.

Going Deeper

- **Functional Core, Imperative Shell** — Gary Bernhardt (Destroy All Software)
- **Domain Modeling Made Functional** — Scott Wlaschin
- **league/pipeline**, **illuminate/collections**, **phpoption/phpoption**
- **Mostly Adequate Guide to FP** — free online (JavaScript, but applies)

Thank you.

// QUESTIONS?

Steal what works. Skip the rest.