DAMIAN JÓŹWIAK

LARAFONY: SOLIDny Framework PHP

TWORZENIE NOWOCZESNEGO SILNIKA W PHP 8.5

SOLIDne fundamenty, wzorce projektowe
i najwyższa wydajność, zależny tylko od PSR

Larafony: SOLIDny Framework PHP

Tworzenie nowoczesnego silinika w PHP8.5

Autor: **Damian Jóźwiak**

Projekt Okładki: **Maciej Pieda - Pieda Art Design**

Wydanie drugie: **2025**

Strona internetowa książki: **https://masterphp.eu**

Kod źródłowy dostępny pod adresem:
https://github.com/DJWeb-Damian-Jozwiak/larafony

Dokumentacja Larafony: **https://larafony.com**

Zastrzeżenie

Framework stworzony w niniejszej książce jest w pełni pokryty testami automatycznymi, co zapewnia, że każda z jego funkcji została dokładnie zweryfikowana pod kątem poprawności działania. Autor książki, bazując na swojej aktualnej wiedzy, uznaje framework za rozwiązanie stabilne i bezpieczne. Należy jednak pamiętać, że publikacja ta powstała w celach edukacyjnych i stanowi bardziej dowód koncepcji (*ang. proof of concept*) tworzenia własnego frameworka, przy czym szczegółowo opisuje działanie najważniejszych jego komponentów wewnętrznych.

Pomimo wysokiej jakości i bezpieczeństwa stworzonego kodu, autor pragnie wyraźnie zaznaczyć, że **nie rekomenduje** korzystania z tego frameworka w rzeczywistych środowiskach produkcyjnych (chociaż na tym frameworku została zbudowana działająca produkcyjnie strona larafony.com stanowiąca dokumentację frameworka - co pozwoliło mi wykryć od groma błędów które przeoczyłem w testach). Framework został opracowany głównie jako materiał dydaktyczny, a jego zastosowanie w warunkach produkcyjnych wiąże się z ryzykiem, które każdy użytkownik podejmuje na własną odpowiedzialność.

PS: Jeśli używane środowisko wymaga użycia MsSQL produkcyjne użycie Larafony jest wysoce niezalecane. Z dwóch powodów:

- w larafony stworzyłem jedynie driver do MySQL (co na potrzeby edukacyjne wystarcza). Być może kiedyś to zmienię (PS: jeśli czujesz się na siłach stwórz PR)
- Powód nr 2 jest dużo bardziej brutalną ścianą zderzenia z rzeczywistością. Na momemnt pisania tego ebooka (listopad/grudzień 2025) nie istnieje oficjalne rozszerzenie php mssql działające w php8.5 (nawet 8.4 ma problemy). Autorowi udało się zmusić php8.5 do współpracy z mssql ale zdecydowanie wykracza to poza zakres tego ebooka.

Dlatego też, autor **nie ponosi odpowiedzialności** za jakiekolwiek ewentualne problemy wynikające z użycia frameworka w środowiskach produkcyjnych, takie jak błędy funkcjonalne, awarie systemowe czy potencjalna utrata danych.

Wstęp

Witaj w książce, która łączy technologię z pasją, a kod z opowieścią. Na kolejnych stronach znajdziesz zarówno praktyczne wskazówki, jak i osobiste refleksje, które towarzyszyły mi podczas tworzenia tego frameworka.

AI - przyjaciel Twój to czy wróg

Żyjemy w ciekawych czasach. Możesz zapytać sztuczną inteligencję o wszystko, a nawet... pogadać z tą książką! Dzięki MCP (ang. Model Context Protocol) od Anthropic opisanemu w jednym z końcowych rozdziałów, ten ebook potrafi stać się Twoim rozmówcą.

Ale AI nie zawsze ma rację.

Czasem odpowiada perfekcyjnie. Czasem halucynuje - czyli mówiąc wprost: zmyśla jak mistrz bajek.

Framework, który za chwilę poznasz, powstawał tak jak tworzy się dobre oprogramowanie: powoli, świadomie, z potknięciami (większość zostawiłem tu w formie notatek autora), refaktoryzacją i małymi zwycięstwami, które tylko programista potrafi docenić.

Pewnie zapytasz, czy korzystałem z AI przy pisaniu tej książki. Tak - ale tylko jako konduktora pociągu, nie maszynisty. To ja trzymam ster, ja piszę kod, ja podejmuję decyzje - AI czasem dopisze literówkę, czasem naprawi styl, czasem podsunie inspirację. Framework jest zero-dependency, i ta książka też. AI to tylko narzędzie - jak debuger, jak dokumentacja, jak kubek kawy obok monitora.

Jeśli coś będzie niejasne - pytaj AI śmiało. Tylko pamiętaj jedno: ta książka używa PHP 8.5, którego AI jeszcze nie zna natywnie. Więc jeśli chatbot twierdzi, że dany kod nie zadziała - najpierw uruchom go lokalnie. Bardzo możliwe, że to Ty masz rację.

Nawiązania literackie

By pokazać, że kodowanie to nie tylko sztuka algorytmów, ale także droga, którą każdy z nas przebywa na swój sposób, w książce znajdziesz dwa niecodzienne wprowadzenia.

Pierwsze, inspirowane hiszpańskim utworem *„La historia de un niño"* autorstwa Porty, oddaje hołd uniwersalnej naturze programowania i osobistej drodze rozwoju. Drugie zaś, w lekkim i humorystycznym tonie, odnosi się do ikonicznej czołówki serialu *„Walker, Texas Ranger"* i dedykowane jest wszystkim tym, którzy z determinacją i uśmiechem stawiają czoła codziennym wyzwaniom programistycznym.

Ta książka to nie tylko przewodnik po frameworku, ale także **„współczesna biblia PHP"** – kompendium nowoczesnych standardów, dobrych praktyk i inspiracji do tworzenia oprogramowania. Na jej stronach znajdziesz wszystko, co sprawia, że PHP pozostaje jednym z najpotężniejszych i najbardziej uniwersalnych języków programowania. Od zgodności z PSR po modularne podejście do kodu, każdy rozdział pokazuje, jak stworzyć i testować narzędzia na miarę XXI wieku.

Obie te inspiracje łączy przekonanie, że za każdą linią kodu kryje się historia, a programowanie to nie tylko narzędzie, ale także sposób wyrażania siebie. Rozpocznijmy tę podróż!

Cuando era solo un niño, el código no entendía,
Las letras y números se mezclaban en mi fantasía.
Cada error me frenaba, cada bug me asustaba,
Pero el sueño crecía en mí, mientras el miedo pasaba.

Frente a la pantalla largas noches, el teclado sonaba,
Paso a paso aprendí a crear, de lo pequeño algo se alzaba.
Ahora el código es mi arte, el mundo espera mi canción,
En cada línea veo el poder, es mi mayor ambición.

Este libro es mi guía, el código mi misión,
MVC, patrones, PSR, construyen mi visión.
Con cada capítulo, más claro es el sendero,
Escribiendo líneas, el futuro es lo que quiero.

Cuando era solo un niño, el código no entendía...

[FIRST VERSE]
In the eyes of new PHP
the unsuspecting Behav'
Had better know the truth from wrong or right;

Rules of Solid and PSR always followed
that "dead"language surely beloved
With the Lone Star of the Ranger shining bright.

[CHORUS]
Asymmetric visibility is upon you
everything is tested like TDD says
when it's that framework don't look behind you
'Cause you're the Ranger, who everyone cares

[SECOND VERSE]
In the heart of a Ranger,
he'll never know the danger;
From old where everything is untyped

our new code is comming
so there ain't no need to wonder any test is green and running
To new where each code line is hyped

Spis treści

Rozdział 1

Wstęp

W tym rozdziale wprowadzimy kluczowe pojęcia, omówimy w skrócie dobre praktyki w inżynierii oprogramowania oraz zaprezentujemy wzorce projektowe, które zostały zastosowane. Przybliżymy również, czym jest PSR, dlaczego odgrywa istotną rolę w nowoczesnych aplikacjach, oraz omówimy wybrane standardy. Na zakończenie rozdziału skonfigurujemy środowisko pracy, przygotowując repozytorium oraz narzędzia do testowania, takie jak phpunit i phpstan.

1.1 Dobre praktyki

W inżynierii oprogramowania wymyślono wiele dobrych praktyk programistycznych. W przypadku tworzenia własnego frameworka szczególnie użyteczne będą zasady SOLID oraz DRY

Zasada DRY ("Don't Repeat Yourself") jest kluczowa w programowaniu obiektowym, ponieważ promuje tworzenie kodu, który jest bardziej czytelny, łatwiejszy w utrzymaniu i mniej podatny na błędy. Unikanie powtórzeń kodu oznacza, że każda informacja lub funkcjonalność jest zdefiniowana tylko raz, co ułatwia wprowadzanie zmian i aktualizacji bez ryzyka niespójności. W kontekście programowania obiektowego, DRY zachęca do wykorzystania mechanizmów takich jak dziedziczenie, kompozycja i polimorfizm, aby efektywnie zarządzać współdzielonymi zachowaniami i właściwościami między obiektami, co prowadzi do bardziej elastycznego i skalowalnego kodu.

1.1.1 SOLID

SOLID jest skrótem oznaczającym 5 zasad projektowania w programowaniu obiektowym wymyślonych przez Roberta C. Martina. Kolejne litery odnoszą się do następujących zasad

- **S** - Single-Responsibility Principle (SRP)
- **O** - Open-closed Principle (OCP)
- **L** - Liskov Substitution Principle (LSP)
- **I** - Interface Segregation Principle (ISP)
- **D** - Dependency Inversion Principle (DIP)

Rozważmy następujące klasy

```php
<?php
declare(strict_types=1);

namespace Chapter1;

interface ShapeContract
{
    public float $area { get; }
}

interface CalculatorContract
{
    public function calculate(): float;
}

interface TextContract
{
    public function text(): string;
}

interface JsonContract
{
```

```
23      public function json(): string;
24  }
25
26  class Square implements ShapeContract
27  {
28      public function __construct(private readonly int
            $length){}
29
30      public float $area {
31          get => $this->length * $this->length;
32      }
33  }
34
35  class Circle implements ShapeContract
36  {
37      public function __construct(private readonly int
            $radius){}
38
39      public float $area {
40          get => $this->radius * $this->radius * pi();
41      }
42  }
43
44  readonly class AreaCalculator implements
        CalculatorContract
45  {
46      /**
47       * @param array<int, ShapeContract> $shapes
48       */
49      public function __construct(protected array $shapes)
50      {
51
52      }
53      public function calculate(): float
54      {
55          return array_map(fn(ShapeContract $shape) =>
                $shape->area, $this->shapes)
56                  |> array_sum(...);
57      }
58  }
59
```

```
60 readonly class BigAreaCalculator extends AreaCalculator
61 {
62     public function calculate(): float
63     {
64
65         $initial = parent::calculate();
66         //some additional logic and filters like not less
               than
67         return $initial;
68     }
69 }
70
71 readonly class SumCalculatorOutputter implements
       TextContract, JsonContract
72 {
73     public function __construct(private CalculatorContract
           $calculator){}
74     public function text(): string
75     {
76         return "The sum of the areas of the shapes is: " .
               $this->calculator->calculate();
77     }
78
79     public function json(): string
80     {
81         return json_encode(['sum' => $this->calculator->
               calculate()]);
82     }
83 }
```

Listing 1.1: Zasady SOLID

Zasada jednej odpowiedzialności - SRP

Klasa powinna mieć tylko jedną odpowiedzialność. Jedyne co potrafią klasy `Square` i `Circle` to policzenie swojego pola. Zadaniem klasy `AreaCalculator` jest wyliczenie pola dla tablicy figur, i bez znaczenia jest tutaj, w jaki sposób każda z figur liczy swoje pole. Natomiast jedynym zadaniem klasy `SumCalculatorOutputter` jest wyświetlenie wyniku jako tekst lub json.

Zasada otwarte-zamknięte - OCP

Klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje. Ponieważ klasy `Square` i `Circle` używają interfejsu do obliczenia swojego pola, dodanie nowej klasy (np trójkąta) nie wymaga od nas modyfikacji klasy kalkulatora. Dodatkowo użycie `array_map` gwarantuje, że wszystkie obiekty będą miały zaimplementowany getter dla pola `$area`.

Zasada podstawienia Liskova - LSP

Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów. Klasa `BigAreaCalculator` spełnia tą zasadę ponieważ nie ma większego znaczenia co dokładnie robi klasa bazowa

Zasada segregacji interfejsów - ISP

Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny. Klasa `SumCalculatorOutputter` spełnia tą zasadę, ponieważ za każdy z rodzajów zwracanego wyjścia odpowiada dedykowany interfejs.

Zasada odwrócenia zależności - DIP

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji. Każdy z modułów spełnia tą zasadę. Kalkulatory w konstruktorze przyjmują interfejs dla kształtu, natomiast klasa do wyświetlania wyników przyjmuje w konstruktorze interfejs kalkulatora. Należy dodatkowo zwrócić uwagę, że wszystkie klasy otrzymują swoje zależności już w konstruktorze. Wykorzystamy to już w rozdziale trzecim gdzie zaimplementujemy kontener Dependency Injection stanowiący podstawę naszego frameworka.

1.1.2 YAGNI

YAGNI to akronim od angielskiego `You Aren't Gonna Need It`. W praktyce oznacza to, że do pliku `composer.json` dodajemy tylko te zależności, które są naprawdę niezbędne. W naszym przypadku będą to:

- Interfejsy PSR,

- biblioteka `Carbon` do zarządzania datami,
- biblioteka `Faker` do generowania przykładowych danych i fabryk modeli,
- biblioteka `Symfony/Mailer` do niskopoziomowej obsługi maili.

Takie podejście pozwala zminimalizować ryzyko potencjalnych luk w bezpieczeństwie oraz ogranicza nadmiarowy kod.

1.2 Wzorce projektowe

Wzorce projektowe są sprawdzonymi rozwiązaniami często występujących problemów w programowaniu obiektowym. Pomagają one w organizacji kodu w sposób, który jest czytelny, elastyczny i łatwy do rozwijania. W tej sekcji przyjrzymy się kilku popularnym wzorcom projektowym, które odgrywają kluczową rolę w tworzeniu skalowalnego i efektywnego oprogramowania.

1.2.1 Budowniczy

Wzorzec Budowniczy (Builder) służy do konstruowania złożonych obiektów krok po kroku. Jest szczególnie przydatny, gdy obiekt posiada wiele opcjonalnych parametrów lub wymaga skomplikowanej inicjalizacji. Dzięki zastosowaniu Budowniczego można oddzielić proces tworzenia obiektu od jego reprezentacji, co ułatwia modyfikacje i dodawanie nowych opcji.

```
1 <?php
2
3 declare(strict_types=1);
4
5 namespace Larafony\Framework\Database\Base\Query;
6
7 use Closure;
8 use Larafony\Framework\Database\Base\Contracts\
    ConnectionContract;
```

```php
use Larafony\Framework\Database\Base\Query\Contracts\
    GrammarContract;
use Larafony\Framework\Database\Base\Query\Contracts\
    QueryBuilderContract;
use Larafony\Framework\Database\Base\Query\Enums\JoinType;
use Larafony\Framework\Database\Base\Query\Enums\
    OrderDirection;

/**
 * Base Query Builder - provides fluent API
 * Abstract class - no SQL building here, only state
     management
 * Concrete implementations in drivers (MySQL, PostgreSQL,
     etc.)
 *
 * @phpstan-consistent-constructor
 */
abstract class QueryBuilder implements
    QueryBuilderContract
{
    protected GrammarContract $grammar;
    protected QueryDefinition $query;

    public function __construct(protected readonly
        ConnectionContract $connection)
    {
    }

    abstract public function table(string $table): static;

    /**
     * @param array<int, string> $columns
     */
    abstract public function select(array $columns):
        static;

    abstract public function where(string $column, string
        $operator, mixed $value): static;

    abstract public function orWhere(string $column,
        string $operator, mixed $value): static;
```

```
40
41      /**
42       * @param array<int, mixed> $values
43       */
44      abstract public function whereIn(string $column, array
            $values): static;
45
46      /**
47       * @param array<int, mixed> $values
48       */
49      abstract public function whereNotIn(string $column,
            array $values): static;
50
51      abstract public function whereNull(string $column):
            static;
52
53      abstract public function whereNotNull(string $column):
             static;
54      abstract public function whereNested(Closure $callback
            , string $boolean): static;
55
56      /**
57       * @param array<int, mixed> $values
58       */
59      abstract public function whereBetween(string $column,
            array $values): static;
60
61      abstract public function whereLike(string $column,
            string $pattern): static;
62
63      abstract public function join(
64          string $table,
65          string|Closure $first,
66          ?string $operator = null,
67          ?string $second = null,
68          JoinType $type = JoinType::INNER,
69      ): static;
70
71      abstract public function leftJoin(
72          string $table,
73          string|Closure $first,
```

```
74          ?string $operator = null,
75          ?string $second = null,
76      ): static;
77
78      abstract public function rightJoin(
79          string $table,
80          string|Closure $first,
81          ?string $operator = null,
82          ?string $second = null,
83      ): static;
84
85      //other abstract methods
86  }
```

Listing 1.2: Przykład wzorca Budowniczy

Przy wszelkiego rodzaju frameworkach ORM (czyli mapowanie obiektowo-relacyjne `Object-relation mapping`) zapytania budowane są niczym z klocków Lego. Powyższa klasa `QueryBuilder` jest tego świetnym przykładem. Jej jedynym zadaniem jest stworzenie zapytania SQL z wykorzystaniem `Fluent Interface`. Klasa ta jest abstrakcyjna ponieważ definiuję ogólną strukturę SQL - konkretna implementacja znajduje się w klasach implementujących połączenie z bazą danych MySQL

1.2.2 Dekorator

Wzorzec Dekorator (Decorator) umożliwia dynamiczne rozszerzanie funkcjonalności obiektów bez potrzeby modyfikacji ich kodu. Dekorator jest często używany, aby dodać nowe zachowania do klasy w sposób elastyczny, bez konieczności korzystania z dziedziczenia. Pozwala to uniknąć nadmiernej ilości podklas i upraszcza utrzymanie kodu. Wzorzec ten jest niezwykle pomocny w osiągnięciu pierwszej zasady SOLID i tym samym złożoności klas wynoszącej 5 lub mniej.

Standardy PSR-7 oraz PSR-17 (dokładniej omówione w kolejnym rozdziale) dość szczegółowo opisują operacje jakie można wykonać na wgrywanym pliku. Dekoratory są tutaj świetnym przykładem. Spójrzmy na poniższe klasy

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Http\UploadedFile\Decorators;

use Psr\Http\Message\StreamInterface;
use Psr\Http\Message\UploadedFileInterface;

abstract class UploadedFileDecorator implements
    UploadedFileInterface
{
    public function __construct(
        protected readonly UploadedFileInterface $file,
    ) {
    }

    public function getStream(): StreamInterface
    {
        return $this->file->getStream();
    }

    public function moveTo(string $targetPath): void
    {
        $this->file->moveTo($targetPath);
    }

    public function getSize(): ?int
    {
        return $this->file->getSize();
    }

    public function getError(): int
    {
        return $this->file->getError();
    }

    public function getClientFilename(): ?string
    {
        return $this->file->getClientFilename();
    }
```

```
41
42      public function getClientMediaType(): ?string
43      {
44          return $this->file->getClientMediaType();
45      }
46  }
```

Listing 1.3: Bazowy dekorator PSR-7

Klasa ta stanowi bazową logikę dla pozostałych jak ten poniżej sprawdzający że plik może być przeniesiony tylko raz

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Http\UploadedFile\Decorators;
6
7   class MoveStatusDecorator extends UploadedFileDecorator
8   {
9       private bool $moved = false;
10
11      public function moveTo(string $targetPath): void
12      {
13          if ($this->moved) {
14              throw new \RuntimeException('Cannot move file;
                    already moved!');
15          }
16
17          parent::moveTo($targetPath);
18          $this->moved = true;
19      }
20  }
```

Listing 1.4: Dekorator MoveStatusDecorator

Użycie tych dekoratów jest wyjątkowo proste i odbywa się przy pomocy wzorca

Fabryka

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Http\Factories;

use Larafony\Framework\Http\UploadedFile;
use Larafony\Framework\Http\UploadedFile\Decorators\
   ErrorValidatorDecorator;
use Larafony\Framework\Http\UploadedFile\Decorators\
   MoveStatusDecorator;
use Larafony\Framework\Http\UploadedFile\Decorators\
   PathValidatorDecorator;
use Psr\Http\Message\StreamInterface;
use Psr\Http\Message\UploadedFileFactoryInterface;
use Psr\Http\Message\UploadedFileInterface;

final readonly class UploadedFileFactory implements
   UploadedFileFactoryInterface
{
    public function createUploadedFile(
        StreamInterface $stream,
        ?int $size = null,
        int $error = \UPLOAD_ERR_OK,
        ?string $clientFilename = null,
        ?string $clientMediaType = null,
    ): UploadedFileInterface {
        return $this->create($stream, $size ?? 0, $error,
           $clientFilename, $clientMediaType);
    }

    public static function create(
        StreamInterface|string $streamOrFile,
        int $size,
        int $error,
        ?string $clientFilename = null,
        ?string $clientMediaType = null,
    ): UploadedFileInterface {
```

```
34          $file = new UploadedFile($streamOrFile, $size,
                $error, $clientFilename, $clientMediaType);
35          $file = new ErrorValidatorDecorator($file);
36          $file = new MoveStatusDecorator($file);
37
38          return new PathValidatorDecorator($file);
39      }
40  }
```

Listing 1.5: klasa UploadedFileFactory

1.2.3 Fabryka

Wzorzec Fabryka (Factory) pozwala na tworzenie obiektów bez bezpośredniego używania operatora new. Zamiast tego, cały proces jest zarządzany przez metodę fabrykującą, co umożliwia elastyczną kontrolę nad tworzonymi instancjami i unikanie ścisłych zależności między klasami.

Pierwszy przykład tego wzorca został pokazany powyżej. Ale fabryki to nie tylko PSR-17. Fabryki są jednym z najczęściej używanych w wzorców w Larafony, ponieważ przy pomocy fabryk i konfiguracji (rozdział 5) możemy tworzyć dowolny sterownik (ang *Driver*) dla dowolnego fragmentu logiki. Poniżej przykład dla Cache z rozdziału 25

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Cache\Factories;
6
7   use Larafony\Framework\Cache\CacheItemPool;
8   use Larafony\Framework\Cache\Contracts\StorageContract;
9   use Larafony\Framework\Config\Contracts\ConfigContract;
10  use Psr\Cache\CacheItemPoolInterface;
11
12  class StorageFactory
```

```
{
    public function __construct(
        private ConfigContract $config,
    ) {
    }

    public function create(string $storeName):
       CacheItemPoolInterface
    {
        $storeConfig = $this->config->get('cache.stores.'
           . $storeName);
        $configArray = $storeConfig instanceof \
           ArrayObject ? $storeConfig->getArrayCopy() : (
           array) $storeConfig;

        $driver = $configArray['driver'] ?? throw new \
           InvalidArgumentException(
            "Cache store '{$storeName}' must have 'driver'
                key in config (file, redis, or memcached)"
        );

        $storage = match ($driver) {
            'file' => $this->createFileStorage(
               $configArray),
            'memcached' => $this->createMemcachedStorage(
               $configArray),
            'redis' => $this->createRedisStorage(
               $configArray),
            default => throw new \InvalidArgumentException
               ("Unsupported cache driver: {$driver}"),
        };
        return new CacheItemPool($storage);
    }
    /**
     * @param array<string, mixed> $config
     */
    private function createFileStorage(array $config):
       StorageContract
    {
        return new FileStorageFactory()->create($config);
    }
```

```
43
44      /**
45       * @param array<string, mixed> $config
46       */
47      private function createMemcachedStorage(array $config)
            : StorageContract
48      {
49          if (! extension_loaded('memcached')) {
50              throw new \RuntimeException('Memcached
                    extension is not loaded');
51          }
52          return new MemcachedStorageFactory()->create(
                $config);
53      }
54
55      /**
56       * @param array<string, mixed> $config
57       */
58      private function createRedisStorage(array $config):
            StorageContract
59      {
60          if (! extension_loaded('redis')) {
61              throw new \RuntimeException('Redis extension
                    is not loaded');
62          }
63          return new RedisStorageFactory()->create($config);
64      }
65  }
```

Listing 1.6: klasa StorageFactory

1.2.4 Fasada

Wzorzec Fasada (Facade) dostarcza uproszczony interfejs do bardziej złożonego systemu klas lub skomplikowanego API. Fasada ukrywa złożoność systemu i umożliwia klientom łatwiejszy dostęp do jego funkcji bez konieczności głębszego zrozumienia wewnętrznych szczegółów. Wzorzec ten będzie najczęściej

wykorzystywany w tworzonym frameworku. Główna klasa danego modułu będzie zazwyczaj fasadą dla wielu innych klas tworzących dany moduł

Bardzo ciekawym przykładem wzorca Fasady jest standard PSR-3 (omówiony w rozdziale 14). Standard ten definiuje jeden interfejs `LoggerInterface` który jest właściwie instrukcją co powinna zawierać każda fasada logująca (czy to do bazy danych, czy plików, czy logów systemowych).

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Log;

use Larafony\Framework\Container\Contracts\
   ContainerContract;
use Larafony\Framework\Web\Application;
use Psr\Log\LoggerInterface;

final readonly class Log
{
    private LoggerInterface $logger;
    public function __construct(private ContainerContract
       $container)
    {
        $this->logger = $this->container->get(
           LoggerInterface::class);
    }


    /**
     * @param string $message
     * @param array<int|string, mixed> $context
     *
     * @return void
     */
    public function emergency(string $message, array
       $context = []): void
    {
        $this->logger->emergency($message, $context);
```

```
29      }
30
31      /**
32       * @param string $message
33       * @param array<int|string, mixed> $context
34       *
35       * @return void
36       */
37      public function alert(string $message, array $context
            = []): void
38      {
39          $this->logger->alert($message, $context);
40      }
41
42      /**
43       * @param string $message
44       * @param array<int|string, mixed> $context
45       *
46       * @return void
47       */
48      public function critical(string $message, array
            $context = []): void
49      {
50          $this->logger->critical($message, $context);
51      }
52
53      /**
54       * @param string $message
55       * @param array<int|string, mixed> $context
56       *
57       * @return void
58       */
59      public function error(string $message, array $context
            = []): void
60      {
61          $this->logger->error($message, $context);
62      }
63
64      /**
65       * @param string $message
66       * @param array<int|string, mixed> $context
```

```
67          *
68          * @return void
69          */
70         public function warning(string $message, array
               $context = []): void
71         {
72             $this->logger->warning($message, $context);
73         }
74
75         /**
76          * @param string $message
77          * @param array<int|string, mixed> $context
78          *
79          * @return void
80          */
81         public function notice(string $message, array $context
               = []): void
82         {
83             $this->logger->notice($message, $context);
84         }
85
86         /**
87          * @param string $message
88          * @param array<int|string, mixed> $context
89          *
90          * @return void
91          */
92         public function info(string $message, array $context =
               []): void
93         {
94             $this->logger->info($message, $context);
95         }
96
97         /**
98          * @param string $message
99          * @param array<int|string, mixed> $context
100         *
101         * @return void
102         */
103        public function debug(string $message, array $context
               = []): void
```

```
104     {
105         $this->logger->debug($message, $context);
106     }
107 }
```

Listing 1.7: Fasada Log

1.3 PSR

PSR (PHP Standard Recommendations) to zestaw zaleceń mających na celu ujednolicenie stylu i praktyk kodowania w ekosystemie PHP. Każdy standard określony w ramach PSR wpływa na różne aspekty tworzenia aplikacji, począwszy od organizacji kodu po bardziej złożone zagadnienia jak zarządzanie żądaniami HTTP czy wykorzystanie wzorców projektowych. PSR jest kluczowy, ponieważ zapewnia spójność i przewidywalność w różnych projektach, ułatwiając współpracę między deweloperami oraz integrację zewnętrznych bibliotek.

Każdy standard PSR jest rozwijany przez PHP-FIG (PHP Framework Interoperability Group) i jest szeroko stosowany w całym ekosystemie PHP. Zrozumienie tych standardów i ich implementacja jest kluczowa dla tworzenia nowoczesnych, skalowalnych aplikacji.

Wszystkie z wymienionych poniżej standardów zostaną wykorzystane w tworzonym frameworku, z wyjątkiem PSR-12, którego będziemy trzymać się na wszystkich listingach

1.3.1 PSR-3, tworzymy logi

PSR-3 definiuje interfejs do tworzenia logów w aplikacjach PHP. Standard ten wprowadza zestaw metod, które mogą być zaimplementowane przez różne biblioteki logowania, co pozwala na wymienne stosowanie różnych implementacji w zależności od potrzeb projektu. Główne metody PSR-3 to np. `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info` oraz `debug`.

Implementacja PSR-3 pozwala na lepsze monitorowanie działania aplikacji, identyfikację potencjalnych problemów oraz analizę działania aplikacji w warunkach produkcyjnych. Dzięki spójności interfejsu możemy łatwo podmienić mechanizm logowania na inny, bez konieczności modyfikacji kodu, co zapewnia większą elastyczność i modularność.

1.3.2 PSR-4 i autoloading

PSR-4 wprowadza standard dla automatycznego ładowania klas (autoloading) w PHP, który jest kluczowy dla efektywnej organizacji kodu w większych projektach. Dzięki PSR-4 każda klasa w projekcie jest mapowana na strukturę katalogów, co pozwala na automatyczne wczytywanie klas bez potrzeby ręcznego `require` lub `include`.

Korzystanie z PSR-4 znacznie ułatwia zarządzanie kodem i wspiera lepszą organizację przestrzeni nazw (`namespace`). Umożliwia to także łatwiejszą współpracę z zewnętrznymi bibliotekami oraz frameworkami, ponieważ wszystkie przestrzenie nazw są jasno zdefiniowane i mogą być ładowane w sposób automatyczny.

W kontekście dużych projektów, takich jak frameworki MVC (Model-View-Controller), stosowanie standardów PSR, a szczególnie PSR-4, jest bardzo ważne. Frameworki MVC są złożone i składają się z wielu komponentów. Standardy takie jak PSR-4 pomagają utrzymać kod w porządku, ułatwiają nawigację i zrozumienie struktury projektu.

1.3.3 PSR-6 cache

PSR-6 to standard opracowany przez PHP-FIG, definiujący interfejsy dla systemów cache'owania w PHP. Jego celem jest ujednolicenie sposobu zarządzania pamięcią podręczną w aplikacjach PHP, niezależnie od konkretnej implementacji.

Główne pojęcia w PSR-6 obejmują:

- **Cache Pool**: Główna jednostka zarządzania pamięcią podręczną, która umożliwia przechowywanie i pobieranie obiektów cache'owanych.

- **Cache Item:** Pojedynczy element pamięci podręcznej, zawierający dane oraz metadane, takie jak czas wygaśnięcia.
- **Tagi i TTL:** PSR-6 pozwala na dodawanie tagów do elementów cache'owanych oraz określanie czasu ich życia (*Time-To-Live*).

1.3.4 PSR-7, requesty HTTP

PSR-7 definiuje standard dotyczący zarządzania żądaniami i odpowiedziami HTTP w aplikacjach PHP. Standard ten wprowadza interfejsy dla klas, które reprezentują zarówno żądania, jak i odpowiedzi HTTP, pozwalając na łatwe manipulowanie nimi podczas przetwarzania w aplikacji. Kluczowe pojęcia związane z PSR-7 to `RequestInterface`, `ResponseInterface`, `StreamInterface` oraz `UriInterface`.

PSR-7 jest często stosowany w ramach aplikacji webowych, szczególnie tych opartych na wzorcu middleware, gdzie każde żądanie jest przetwarzane na różnych poziomach. Dzięki standaryzacji, aplikacje mogą bez problemu integrować różne biblioteki, które również implementują ten standard.

1.3.5 PSR-11, dodajemy kontener zależności

PSR-11 definiuje interfejs dla kontenerów zależności (Dependency Injection Containers). Kontener zależności to kluczowy element architektury wielu aplikacji PHP, który umożliwia efektywne zarządzanie zależnościami między klasami. Dzięki PSR-11 różne kontenery zależności mogą być wymiennie stosowane w aplikacjach, pod warunkiem, że implementują wymagane interfejsy.

Użycie PSR-11 sprawia, że kod staje się bardziej modularny i testowalny. Łatwość zarządzania zależnościami sprawia, że można dynamicznie dostarczać różne implementacje klas, co zwiększa elastyczność projektu i pozwala na lepszą separację odpowiedzialności.

1.3.6 PSR-12 czyli jakość kodu

PSR-12 rozszerza i doprecyzowuje wcześniejsze zalecenia dotyczące stylu kodowania w PHP, opierając się na standardach wprowadzonych przez PSR-1 i PSR-2. Określa on szczegółowe wytyczne dotyczące formatowania kodu, takie jak użycie odstępów, strukturę deklaracji klas i funkcji, a także formatowanie bloków kontrolnych.

Wprowadzenie PSR-12 ma na celu poprawę czytelności kodu oraz ułatwienie jego przeglądania przez różnych deweloperów. Zgodność z tym standardem sprawia, że kod jest bardziej spójny, co w długiej perspektywie poprawia jego utrzymywalność.

1.3.7 PSR-14, event dispatcher

PSR-14 definiuje interfejsy dla mechanizmu eventów, umożliwiając aplikacjom PHP korzystanie z wzorca projektowego Observer. Standard ten określa, w jaki sposób można rejestrować, wysyłać i przetwarzać zdarzenia w aplikacji. Mechanizm eventów pozwala na lepsze zarządzanie komunikacją wewnętrzną w aplikacji, szczególnie w przypadku skomplikowanych projektów, gdzie różne moduły muszą reagować na określone zdarzenia.

Stosowanie PSR-14 pozwala na luźne powiązanie między komponentami aplikacji, co ułatwia wprowadzanie nowych funkcji i modyfikacje bez naruszania struktury istniejącego kodu.

1.3.8 PSR-15, dodajemy middleware

PSR-15 wprowadza standard dotyczący middleware w aplikacjach PHP. Middleware to warstwy, które pośredniczą w przetwarzaniu żądań HTTP, umożliwiając np. autoryzację, walidację danych czy modyfikację żądania. PSR-15 definiuje interfejsy dla klas middleware, co pozwala na tworzenie spójnych i łatwo rozszerzalnych rozwiązań.

Dzięki implementacji PSR-15 możliwe jest tworzenie modułowych aplikacji, w których poszczególne warstwy logiki mogą być dodawane lub modyfikowane

bez potrzeby ingerencji w cały kod aplikacji. Middleware to kluczowy element w nowoczesnych frameworkach PHP, takich jak Symfony czy Laravel.

1.3.9 PSR-17, HTTP factory

PSR-17 definiuje interfejsy dla fabryk (factory) tworzących obiekty związane z HTTP, takie jak żądania (`RequestFactoryInterface`), odpowiedzi (`ResponseFactoryInterface`) i strumienie (`StreamFactoryInterface`). Dzięki temu standardowi możliwe jest tworzenie obiektów HTTP w sposób spójny i elastyczny, co jest szczególnie przydatne w dużych aplikacjach, które muszą dynamicznie reagować na różne rodzaje żądań.

PSR-17 pozwala na unifikację procesu tworzenia obiektów HTTP, co zwiększa interoperacyjność między różnymi bibliotekami i frameworkami. To podejście sprzyja budowaniu elastycznych i skalowalnych rozwiązań webowych.

1.3.10 PSR-20, Clock

PSR-20 wprowadza standaryzowany interfejs zegara (`ClockInterface`), którego zadaniem jest dostarczanie aktualnego czasu w sposób kontrolowany oraz testowalny. Zegar systemowy nie jest już traktowany jako globalny zasób dostępny poprzez funkcje takie jak `time()` czy `new DateTimeImmutable()`, lecz jako zależność przekazywana do klas, które tego czasu potrzebują.

To podejście znacząco poprawia testowalność kodu. Dzięki wykorzystaniu kontraktu zegara możemy w prosty sposób zamienić rzeczywisty czas na zegar testowy (np. zamrożony lub manipulowany), co pozwala na deterministyczne testowanie operacji zależnych od czasu. Zamiast polegać na nieprzewidywalnych wartościach systemowych, aplikacja korzysta z zegara, który można kontrolować.

PSR-20 eliminuje również powtarzający się problem „ukrytej zależności systemu" — czasu, który dotychczas był pobierany w różnych miejscach aplikacji bez możliwości jego ujednolicenia. Dzięki temu standardowi klasy stają się bardziej spójne, a logika czasu jest wyraźnie oddzielona od logiki biznesowej.

Obecnie najpopularniejsza biblioteką implementującą ten standard jest `Carbon` i integracja z tą biblioteką zostanie pokazana przy okazji innych mostów

w końcowej części książki. Natomiast już w rodziale 3 zostania pokazana edukcyjna, lecz wystarczająca na potrzeby niniejszego ebooka, implementacja PSR-20 wykorzystywana konsekwentnie w całym frameworku.

1.4 Zarządzanie zależnościami

Composer jest standardowym narzędziem do zarządzania zależnościami w PHP. Umożliwia on deklarowanie bibliotek wymaganych przez projekt, instalowanie ich, aktualizowanie oraz kontrolowanie zgodności wersji. W ostatnich latach Composer rozwinął się również w kierunku narzędzia zapewniającego bezpieczeństwo środowiska pracy.

Od wersji 2.4 (2022), Composer włączył mechanizmy znane wcześniej z pakietu `roave/security-advisories` do polecenia `composer audit`, które pozwala wykrywać znane podatności w zainstalowanych bibliotekach.

Jednak dopiero wersja 2.9 - wymagana przez PHP 8.5 i wydana 13 listopada 2025 roku — wprowadziła najbardziej rygorystyczną zmianę: **domyślne blokowanie instalacji pakietów posiadających znane podatności bezpieczeństwa (CVE[1])**. Oznacza to, że uruchomienie polecenia `composer install` może zostać przerwane, jeśli którakolwiek z zależności w projekcie znajduje się na liście zagrożeń.

Jest to ogromny krok naprzód w kierunku bezpieczeństwa ekosystemu PHP i znacząco podnosi poziom kontroli nad wykorzystywanymi bibliotekami. Można śmiało powiedzieć, że od końca 2025 roku Composer pełni rolę nie tylko menedżera pakietów, lecz także warstwy bezpieczeństwa dla całych projektów.

1.4.1 Zarządzanie zależnościami

Composer nie jest menedżerem pakietów w tradycyjnym sensie, takim jak Yum czy Apt. Zamiast tego zarządza bibliotekami na poziomie konkretnego projektu, instalując je w katalogu (np. `vendor`) wewnątrz projektu. Domyślnie nic nie jest instalowane globalnie, co czyni Composer menedżerem zależności. Jed-

[1] Common Vulnerabilities and Exposures

nak dla wygody dostępna jest możliwość instalacji globalnej za pomocą komendy `global`.

Idea ta nie jest nowa i Composer czerpie silne inspiracje z narzędzi takich jak npm dla Node.js czy Bundler dla Ruby.

1.4.2 Przykład użycia

Załóżmy, że masz projekt, który zależy od wielu bibliotek. Niektóre z tych bibliotek również mają swoje zależności.

Composer umożliwia:

- deklarowanie bibliotek, od których zależy projekt,
- określenie, które wersje pakietów mogą i powinny zostać zainstalowane, a następnie ich instalację (czyli pobieranie do projektu),
- aktualizację wszystkich zależności jednym poleceniem.

Więcej informacji można znaleźć na stronie: `https://getcomposer.org/doc/00-intro.md`

1.5 Konfiguracja lokalnego repozytorium

Nota autora

WAŻNE!
Poniższa sekcja bazuje na pierwszej wersji kursu. W kodzie, który znajdziesz na GitHubie, implementacja może minimalnie różnić się od prezentowanej tutaj. Wynika to z faktu, że aplikacja demonstracyjna została wydzielona do osobnego repozytorium (co nastąpi formalnie w rozdziale 18).
Na potrzeby tej książki nie ma to jednak większego znaczenia. Kluczowa jest jedynie struktura: **framework znajduje się w jednym folderze, a przykładowa aplikacja — wraz z plikiem `public/index.php` -**

w drugim. Jest to istotne, ponieważ w kolejnych rozdziałach do katalogu aplikacji będą stopniowo dodawane nowe elementy, takie jak konfiguracja czy migracje bazy danych.
Autor zakłada - z pełnym optymizmem — że w erze wszechobecnych chatbotów AI (bez wymieniania nazw, aby przypadkiem nikomu nie robić reklamy), Drogi Czytelniku, **poradzisz sobie bez problemu!**

1.5.1 Utworzenie struktury katalogów

Zacznijmy od utworzenia podstawowej struktury katalogów projektu. W konsoli (lub terminalu) wykonaj polecenie:

```
mkdir -p my_project/{public,app,framework}
```

1.5.2 Utworzenie głównego pliku projektu

Przejdź do katalogu `public` i utwórz plik `index.php`:

```
touch my_project/public/index.php
```

Możesz edytować ten plik i dodać podstawowy kod PHP, np.:

```
<?php
phpinfo();
```

1.5.3 Utworzenie pustego repozytorium w katalogu framework

W katalogu `framework` utwórz puste repozytorium:

```
cd my_project/framework
git init
```

1.5.4 Dodanie pliku `composer.json`

W katalogu `framework` utwórz plik `composer.json`:

```
touch composer.json
```

W pliku `composer.json` zdefiniuj minimalną strukturę:

```
{
    "name": "my_project/framework",
    "description": "Framework for my PHP project",
    "autoload": {
        "psr-4": {
            "Framework\\": "src/"
        }
    }
}
```

1.5.5 Dodanie pakietu `symfony/var-dumper`

Paczka `symfony/var-dumper` oferuje znacznie bardziej czytelne i wygodne wyświetlanie informacji o zmiennych w porównaniu ze standardowymi funkcjami `var_dump` i `print_r`. Obie te funkcje wyświetlają dane bez formatowania, co w przypadku obiektów czy wielowymiarowych tablic szybko prowadzi do chaosu. Natomiast `VarDumper` renderuje struktury z kolorowaniem składni, czytelnymi wcięciami i interaktywnymi elementami, co znacząco ułatwia debugowanie.

Przejdź teraz do głównego katalogu projektu i utwórz plik `composer.json`:

Nota autora

WAŻNE - Uwaga nr 1
Choć paczki VarDumper sam często używałem (jest wielokrotnie wygodniejsza niż klasyczny `var_dump`), nie została ona umieszczona w kodzie samego frameworka. Natomiast w repozytorium aplikacji demonstracyjnej pakiet ten znajduje się już w sekcji `require-dev`.
WAŻNE - Uwaga nr 2
Tworząc pierwszą edycję kursu błędnie założyłem, że sam `dump()` z paczki Symfony wystarczy do debugowania błędów. Nic bardziej mylnego.
Dlatego tuż po niniejszym wstępie (w kolejnym rozdziale) przedstawiony zostanie prosty, ale wyjątkowo skuteczny handler błędów, który w dalszej części książki zostanie rozbudowany do interaktywnego debuggera - zarówno w środowisku webowym, jak i konsolowym.
Może Cię, Drogi Czytelniku, zdziwić pojawienie się w kodzie „tras" celowo rzucających błędem. Jest to mój własny mechanizm testowy, który zachowałem nawet po kolejnych refaktorach. Nie jest on wymagany (i w odpowiednim momencie zostanie usunięty), ale gorąco zachęcam, aby go na razie pozostawić — jest bardzo pomocny, szczególnie podczas eksperymentów, do których szczerze zachęcam.

```
1 cd ..
2 touch composer.json
```

Dodaj do pliku następującą zawartość, aby dodać lokalne repozytorium frameworka oraz pakiet `symfony/var-dumper`:

```
{
    "repositories": [
        {
            "type": "path",
            "url": "./framework"
        }
    ],
    "require-dev": {
        "my_project/framework": "*",
        "symfony/var-dumper": "^7.4"
    }
}
```

1.5.6 Instalacja zależności

Aby zainstalować zależności, uruchom w terminalu:

```
composer install
```

To polecenie pobierze wszystkie wymagane pakiety i zainstaluje je w katalogu `vendor`.

1.5.7 Konfiguracja vHosta

Konfiguracja wirtualnych hostów (vHostów) jest kluczowa dla hostowania wielu stron na jednym serwerze. Poniżej przedstawiam przykładowe pliki konfiguracyjne dla Apache i Nginx. Upewnij się, że ścieżka podana w plikach odpowiada

miejscu, gdzie zostały utworzone foldery w krokach powyżej. Przed rozpoczęciem konfiguracji, upewnij się, że PHP 8.4 FPM jest zainstalowane:

```
sudo apt update
sudo apt install php8.5-fpm
```

dodaj do pliku `/etc/hosts`

```
127.0.0.1 framework.local
```

Apache

Stwórz plik konfiguracyjny dla vHosta:

```
sudo touch /etc/apache2/sites-available/framework.local.conf
```

Dodaj następującą konfigurację:

```
<VirtualHost *:80>
    ServerName framework.local
    DocumentRoot /var/www/framework/public
    <Directory /var/www/framework/public>
        Options Indexes FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>
    <FilesMatch \.php$>
        SetHandler "proxy:unix:/run/php/php8.5-fpm.sock|fcgi://localhost/"
```

```
    </FilesMatch>
    ErrorLog ${APACHE_LOG_DIR}/framework.local_error.log
    CustomLog ${APACHE_LOG_DIR}/framework.local_access.log
        combined
</VirtualHost>
```

Aktywuj vHosta i przeładuj Apache:

```
sudo a2ensite framework.local.conf
sudo systemctl reload apache2
```

Nginx Stwórz plik konfiguracyjny dla vHosta:

```
sudo touch /etc/nginx/sites-available/framework.local
```

Dodaj następującą konfigurację:

```
server {
    listen 80;
    server_name framework.local;
    root /var/www/framework/public;
    index index.php index.html index.htm;
    location / {
        try_files $uri $uri/ =404;
    }
    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/run/php/php8.5-fpm.sock;
    }
    error_log /var/log/nginx/framework.local_error.log;
    access_log /var/log/nginx/framework.local_access.log;
```

```
15 }
```

Aktywuj vHosta i przeładuj Nginx:

```
1 sudo ln -s /etc/nginx/sites-available/framework.local /etc
     /nginx/sites-enabled/
2 sudo systemctl reload nginx
```

1.6 Punkt wejścia aplikacji

Konfiguracja pokazana w tym rozdziale zakłada jeden punkt wejścia aplikacji - plik `public/index.php`

Współczesne aplikacje internetowe często korzystają z jednego punktu wejściowego, zazwyczaj pliku `public/index.php`. To podejście, popularne w wielu frameworkach takich jak Laravel, Symfony czy Zend, ma wiele zalet, które przyczyniają się do jego powszechnego stosowania.

1.6.1 Zalety Jednego Entry Point

Pierwszą i najważniejszą zaletą jednego punktu wejściowego jest centralizacja zarządzania ruchem sieciowym. Cały ruch przychodzący jest kierowany do jednego pliku, co umożliwia łatwe zarządzanie routingiem, autoryzacją i innymi aspektami aplikacji. Dzięki temu, można w sposób efektywny kontrolować dostęp do różnych zasobów i zapewnić spójność w obsłudze żądań.

Drugą istotną zaletą jest bezpieczeństwo. Posiadanie jednego punktu wejściowego pozwala na implementację globalnych mechanizmów bezpieczeństwa, takich jak filtrowanie wejść, ochrona przed atakami typu SQL Injection czy XSS. Dodatkowo, ułatwia to wdrażanie polityk bezpieczeństwa na poziomie serwera,

takich jak ograniczenie dostępu do określonych plików czy katalogów.

1.7 PHPUnit

PHPUnit to jeden z najpopularniejszych frameworków do testowania jednostkowego w PHP. Umożliwia on pisanie i uruchamianie testów jednostkowych, które pomagają w zapewnieniu, że poszczególne części aplikacji działają zgodnie z oczekiwaniami.

Podstawowe założenia testowania:

- **Arrange (Przygotowanie):** Przygotowanie środowiska testowego, w tym tworzenie obiektów i inicjalizacja danych.
- **Act (Działanie):** Wykonanie operacji, która ma być testowana.
- **Assert (Sprawdzenie):** Weryfikacja, czy wynik operacji jest zgodny z oczekiwaniami.

1.8 phpstan

phpstan to narzędzie do analizy statycznej kodu PHP, które pomaga wykrywać błędy bez potrzeby uruchamiania kodu. phpstan analizuje kod na różnych poziomach szczegółowości, od 0 do 9, gdzie każdy kolejny poziom jest bardziej restrykcyjny i dokładny.

Poziomy phpstan:

- **Poziom 0:** Podstawowa analiza, wykrywanie najprostszych błędów.
- **Poziom 1-4:** Stopniowe zwiększanie restrykcyjności, sprawdzanie typów zmiennych, zgodności metod, itp.
- **Poziom 5-8:** Bardziej szczegółowa analiza, weryfikacja bardziej zaawansowanych przypadków użycia.

- **Poziom 9:** Bardzo restrykcyjna analiza, pełna kontrola typów i zgodności kodu.
- **dostępne od PHPStan 2 Poziom 10:** jeszcze bardziej rygorystyczne podejście do typu mixed — zgłasza błędy nawet dla niejawnych przypadków typu mixed (gdy brak jest określonego typu), a nie tylko dla jawnie zadeklarowanego mixed.

1.9 Instalacja i konfiguracja PHPUnit i phpstan

Aby dodać PHPUnit i phpstan do projektu, należy wykonać następujące kroki:
Krok 1: Instalacja PHPUnit

```
composer require --dev phpunit/phpunit
```

Krok 2: Instalacja phpstan

```
composer require --dev phpstan/phpstan
```

Krok 3: Konfiguracja autoloadingu W pliku ‘composer.json‘ należy dodać sekcję ‘autoload‘ i ‘autoload-dev‘:

```
{
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    },
    "autoload-dev": {
```

```
        "psr-4": {
            "Tests\\": "tests/"
        }
    }
}
```

Następnie należy zaktualizować autoloading:

```
composer dump-autoload
```

Krok 4: Konfiguracja phpstan Utwórz plik ‘phpstan.neon‘ w głównym katalogu projektu:

```
parameters:
    level: max
    paths:
        - src
        - tests
```

Krok 5: Konfiguracja PHPUnit Utwórz plik ‘phpunit.xml‘ w głównym katalogu projektu:

```
<phpunit bootstrap="vendor/autoload.php">
    <testsuites>
        <testsuite name="Unit Tests">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

1.9.1 Pokrycie kodu

Xdebug to rozszerzenie PHP umożliwiające zaawansowane debugowanie, profilowanie oraz generowanie raportów pokrycia kodu. W kontekście testów jednostkowych Xdebug jest często wykorzystywany do analizy tego, które fragmenty kodu zostały faktycznie wykonane podczas testów — co jest kluczowe przy ocenie jakości testów i wykrywaniu nieprzetestowanej logiki.

Nota autora

UWAGA - ważne
W repozytorium towarzyszącym frameworkowi znajdziesz również drugie rozszerzenie służące do pokrycia kodu - `pcov`. W czasie tworzenia frameworka (oraz niniejszego ebooka) PHP 8.5 znajdowało się jeszcze w fazie RC, a `pcov` — ze względu na swoją prostotę - był znacznie łatwiejszy do samodzielnego skompilowania ze źródeł.
Obecnie, Drogi Czytelniku, masz pełną dowolność wyboru rozszerzenia. Oba działają poprawnie i są wspierane przez PHPUnit.

Instalacja Xdebug

Aby zainstalować Xdebug, wykonaj:

```
pecl install xdebug
```

Następnie dodaj wpis do pliku `php.ini`:

```
zend_extension="xdebug"
```

Generowanie raportu pokrycia kodu

Aby wygenerować raport pokrycia kodu w formacie HTML, uruchom PHPUnit z odpowiednią opcją:

```
phpunit --coverage-html coverage
```

Wygenerowany raport znajdziesz w katalogu `coverage`.

Ułatwienie pracy dzięki skryptom Composer

Aby znacznie uprościć codzienną pracę z testami, analizą statyczną oraz narzędziami jakości kodowania, gorąco zachęcam do umieszczenia w pliku `composer.json` następującej sekcji:

```
"scripts": {
    "test": "php8.5 vendor/bin/phpunit --no-coverage",
    "test-coverage": "php8.5 vendor/bin/phpunit --coverage-text",
    "test-coverage-html": "php8.5 vendor/bin/phpunit --coverage-html coverage",

    "analyse": "php8.5 vendor/bin/phpstan analyse --memory-limit=512M",

    "insights": "php8.5 vendor/bin/phpinsights",
    "insights:fix": "php8.5 vendor/bin/phpinsights --fix",
    "insights:quality":
        "php8.5 vendor/bin/phpinsights --min-quality=95 --min-complexity=95 "
        "--min-architecture=95 --min-style=95 --no-interaction",

    "quality": [
        "@analyse",
```

```
            "@test",
            "@insights:quality"
        ]
},
```

Dzięki temu wszystkie kluczowe narzędzia jakości można uruchomić jednym poleceniem:

```
composer quality
```

Co — nie ukrywajmy - jest wyjątkowo wygodne

Nota autora

PS. Ustawienie jakości na poziomie `95%` jest przeznaczone raczej dla entuzjastów (takich jak autor), którzy chcą zagwarantować najwyższą jakość kodu w kursie, który właśnie czytasz. W rzeczywistych projektach — szczególnie tych większych — poziom `70-80%` jest najczęściej w zupełności wystarczający i pozwala zachować rozsądny balans pomiędzy jakością a produktywnością zespołu.
PS2. Jeśli pracujesz w środowisku Windows, istnieje duże prawdopodobieństwo, że będziesz mieć zakończenia linii w formacie `CRLF`. Narzędzie `phpinsights` jest na to dość wrażliwe (co wynika z PSR-12 "All PHP files MUST use the Unix LF (linefeed) line ending only."), szczególnie w sekcji `style`. Masz wówczas trzy możliwości:

1. zignorować wskaźnik stylu,
2. wymusić format `LF` w ustawieniach edytora,
3. uruchomić `insights:fix`, który automatycznie naprawi zakończenia linii.

W praktyce trzecia opcja działa najwygodniej.

Rozdział 5

Dependency Injection

W poprzednich rozdziałach utworzyliśmy system czasu i obsługę HTTP. Każdy z tych komponentów działa samodzielnie, ale w prawdziwej aplikacji muszą ze sobą współpracować. Tu wkracza Dependency Injection – wzorzec projektowy, który pozwala elegancko zarządzać zależnościami między obiektami.

W tym rozdziale zbudujemy kontener DI zgodny z PSR-11, wyposażony w:

- Automatyczne rozwiązywanie zależności (autowire)
- Wsparcie dla dot-notation przy konfiguracji
- Wzorzec Service Provider do modularnej rejestracji usług
- Wyjątki zgodne z PSR-11

Nota autora

W wielu kursach kontener DI pojawia się dopiero w połowie książki. W Larafony zaczynamy od niego wcześnie, bo bez kontenera każdy kolejny komponent wymagałby ręcznego "sklejania" zależności. A to szybko zamienia się w kod wyglądający jak talerz spaghetti – smaczny, ale trudny do utrzymania.

5.1 Dlaczego Dependency Injection jest ważne?

Dependency Injection to nie tylko buzzword do CV – przynosi realne korzyści:

- **Testowalność:** Zależności można łatwo podmienić na mocki lub stuby podczas testów. Zamiast walczyć z globalnym stanem, wstrzykujemy dokładnie to, czego potrzebujemy.
- **Modularność:** Klasy nie są sztywno powiązane z konkretnymi implementacjami. Dziś używasz MySQL, jutro PostgreSQL – zmiana to kwestia konfiguracji, nie refaktoryzacji.
- **Łatwość utrzymania:** Zmiana implementacji zależności nie wymaga modyfikacji klas, które z niej korzystają. Zasada Open/Closed w praktyce.
- **Zarządzanie cyklem życia:** Kontener wie, kiedy tworzyć nowe instancje, a kiedy zwracać istniejące (singleton pattern).

Nota autora

Lubię myśleć o kontenerze DI jak o kelnerze w restauracji. Zamawiasz "kawę" – nie interesuje cię, czy barista użył ekspresu ciśnieniowego czy french pressa. Dostajesz kawę. Kontener działa tak samo: prosisz o `DatabaseInterface`, dostajesz działającą bazę danych. Szczegóły implementacji zostają w kuchni.

5.2 PSR-11 – Container Interface

PSR-11 to standard definiujący interfejs kontenera DI. Jest minimalistyczny – wymaga tylko dwóch metod:

```
1 interface ContainerInterface {
2     public function get(string $id): mixed;
3     public function has(string $id): bool;
4 }
```

Dzięki temu każda biblioteka zgodna z PSR-11 może używać dowolnego kontenera – czy to naszego Larafony, czy PHP-DI, Symfony Container lub innego.

5.3 Architektura kontenera

Nasz kontener składa się z następujących komponentów:

- `ContainerContract` – interfejs rozszerzający PSR-11 o dodatkowe metody
- `Container` – główna klasa kontenera
- `Autowire` – mechanizm automatycznego rozwiązywania zależności
- `ReflectionResolver` – wrapper na PHP Reflection API
- `DotContainer` – storage z obsługą dot-notation
- `ServiceProvider` – bazowa klasa dla modularnej rejestracji usług

5.4 Interfejs kontenera

Rozszerzamy PSR-11 `ContainerInterface` o metody przydatne w codziennej pracy:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Contracts;

use Psr\Container\ContainerInterface;

interface ContainerContract extends ContainerInterface
{
```

```
11      /**
12       * Bind a simple value to the container.
13       */
14      public function bind(string $key, string|int|float|
            bool|null $value): void;
15
16      #[\NoDiscard]
17      public function getBinding(string $key): string|int|
            float|bool|null;
18
19      public function set(string $key, mixed $value): self;
20  }
```

Listing 5.1: Interfejs ContainerContract

Metoda `bind()` służy do przechowywania prostych wartości skalarnych (konfiguracja), podczas gdy `set()` obsługuje pełne obiekty i klasy.

5.5 Wyjątki zgodne z PSR-11

PSR-11 definiuje dwa rodzaje wyjątków:

- `ContainerExceptionInterface` – ogólny błąd kontenera
- `NotFoundExceptionInterface` – żądany element nie istnieje

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Larafony\Framework\Container\Exceptions;
6
7  use Psr\Container\ContainerExceptionInterface;
8
```

```
9  class ContainerError extends \RuntimeException implements
       ContainerExceptionInterface
10 {
11 }
```

Listing 5.2: Klasa ContainerError

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Larafony\Framework\Container\Exceptions;
6
7  use Larafony\Framework\Core\Exceptions\NotFoundError as
       CoreNotFoundError;
8  use Psr\Container\NotFoundExceptionInterface;
9
10 class NotFoundError extends CoreNotFoundError implements
       NotFoundExceptionInterface
11 {
12 }
```

Listing 5.3: Klasa NotFoundError

5.6 Reflection API – zaglądanie pod maskę

Refleksja (Reflection) w PHP to mechanizm pozwalający na analizowanie klas, obiektów oraz ich właściwości w trakcie działania programu. Dzięki `ReflectionClass` możemy dynamicznie uzyskiwać informacje o konstruktorach, metodach i zależnościach obiektów.

Nota autora

Refleksja to jak rentgen dla kodu – pozwala zajrzeć do środka klasy bez jej otwierania. Przydatne, gdy chcesz wiedzieć, czego klasa potrzebuje do życia, zanim ją utworzysz.

5.6.1 Interfejs ReflectionResolver

Aby nie korzystać bezpośrednio z API refleksji w wielu miejscach, tworzymy wrapper:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Contracts;

use Larafony\Framework\Container\Exceptions\ContainerError;
use ReflectionParameter;

interface ReflectionResolverContract
{
    /**
     * Get the constructor parameters for a given class.
     *
     * @param class-string $className The fully qualified class name
     *
     * @return array<int, ReflectionParameter> An array of constructor parameters
     */
    public function getConstructorParameters(string $className): array;

    /**
     * Get the type of a parameter.
     *
```

```
24          * @param ReflectionParameter $parameter The parameter
                to analyze
25          *
26          * @return string|null The type of the parameter or
                null if it can't be determined
27          */
28         public function getParameterType(ReflectionParameter
               $parameter): ?string;
29
30         /**
31          * Check if a parameter has a default value.
32          *
33          * @param ReflectionParameter $parameter The parameter
                to check
34          *
35          * @return bool True if the parameter has a default
                value, false otherwise
36          */
37         public function hasDefaultValue(ReflectionParameter
               $parameter): bool;
38
39         /**
40          * Get the default value of a parameter.
41          *
42          * @param ReflectionParameter $parameter The parameter
                to get the default value from
43          *
44          * @return mixed The default value of the parameter
45          *
46          * @throws ContainerError If the parameter has no
                default value
47          */
48         public function getDefaultValue(ReflectionParameter
               $parameter): mixed;
49
50         /**
51          * Check if a parameter allows null.
52          *
53          * @param ReflectionParameter $parameter The parameter
                to check
54          *
```

```
55          * @return bool True if the parameter allows null,
                false otherwise
56          */
57         public function allowsNull(ReflectionParameter
               $parameter): bool;
58
59         /**
60          * Get the default value for a built-in type.
61          *
62          * @param string $type The built-in type
63          *
64          * @return mixed The default value for the type
65          */
66         public function getDefaultValueForBuiltInType(string
               $type): mixed;
67     }
```

Listing 5.4: Interfejs ReflectionResolverContract

5.6.2 Implementacja ReflectionResolver

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Larafony\Framework\Container\Resolvers;
6
7  use Larafony\Framework\Container\Contracts\
       ReflectionResolverContract;
8  use Larafony\Framework\Container\Exceptions\ContainerError
       ;
9  use ReflectionClass;
10 use ReflectionException;
11 use ReflectionNamedType;
12 use ReflectionParameter;
13
```

```php
class ReflectionResolver implements ReflectionResolverContract
{
    /**
     * Get the constructor parameters for a given class.
     *
     * @param class-string $className The fully qualified class name
     *
     * @return array<int, ReflectionParameter> An array of constructor parameters
     *
     * @throws ReflectionException If the class does not exist or has no constructor
     */
    public function getConstructorParameters(string $className): array
    {
        $reflectionClass = new ReflectionClass($className);
        $constructor = $reflectionClass->getConstructor();

        return $constructor?->getParameters() ?? [];
    }

    /**
     * Get the type of a parameter.
     *
     * @param ReflectionParameter $parameter The parameter to analyze
     *
     * @return string|null The type of the parameter or null if it can't be determined
     */
    public function getParameterType(ReflectionParameter $parameter): ?string
    {
        $type = $parameter->getType();
        return $type instanceof ReflectionNamedType ? $type->getName() : null;
    }
```

```
45
46      /**
47       * Check if a parameter has a default value.
48       *
49       * @param ReflectionParameter $parameter The parameter
             to check
50       *
51       * @return bool True if the parameter has a default
            value, false otherwise
52       */
53      public function hasDefaultValue(ReflectionParameter
            $parameter): bool
54      {
55          return $parameter->isDefaultValueAvailable();
56      }
57
58      /**
59       * Get the default value of a parameter.
60       *
61       * @param ReflectionParameter $parameter The parameter
             to get the default value from
62       *
63       * @return mixed The default value of the parameter
64       *
65       * @throws ContainerError|ReflectionException If the
            parameter has no default value
66       */
67      public function getDefaultValue(ReflectionParameter
            $parameter): mixed
68      {
69          if (! $this->hasDefaultValue($parameter)) {
70              throw new ContainerError("Parameter {
                    $parameter->getName()} has no default value
                    .");
71          }
72
73          return $parameter->getDefaultValue();
74      }
75
76      /**
77       * Check if a parameter allows null.
```

```
78          *
79          * @param ReflectionParameter $parameter The parameter
                to check
80          *
81          * @return bool True if the parameter allows null,
                false otherwise
82          */
83         public function allowsNull(ReflectionParameter
               $parameter): bool
84         {
85             return $parameter->allowsNull();
86         }
87
88         /**
89          * Get the default value for a built-in type.
90          *
91          * @param string $type The built-in type
92          *
93          * @return mixed The default value for the type
94          */
95         public function getDefaultValueForBuiltInType(string
               $type): mixed
96         {
97             return match ($type) {
98                 'int' => 0,
99                 'float' => 0.0,
100                'string' => '',
101                'bool' => false,
102                'array' => [],
103                default => null,
104            };
105        }
106    }
```

Listing 5.5: Klasa ReflectionResolver

5.6.3 Ewolucja typów w PHP

W PHP 5 parametry funkcji nie miały typów – programista musiał zaufać, że dostanie odpowiednie dane. PHP 7.1 wprowadził typowanie skalarne (`int`, `string`, `float`, `bool`), PHP 8.0 dodał typy unii (`int|string`), a PHP 8.2 typy DNF (Disjunctive Normal Form).

Użyty w metodzie `getParameterType()` typ `ReflectionNamedType` pozwala na bezpieczne pobranie nazwy typu parametru.

5.6.4 Nullable types – historia i teraźniejszość

Od PHP 5.1 możliwe było deklarowanie typów z domyślną wartością `null`:

```
function test(string $test = null) {}
test('PHP');  // Dozwolone
test(null);   // Dozwolone
```

PHP 8.0 wprowadził Union Types, umożliwiając jawną deklarację `string|null`. W PHP 8.4 wycofano niejawne typy nullable – deklaracje bez `null` w typie, ale z domyślną wartością `null`, generują ostrzeżenie:

```
function test(array $value = null) {}
test(null);  // Deprecated: Implicit nullable parameter
```

Nota autora

PHP ewoluuje w kierunku jawności typów. To dobra wiadomość – mniej "magii" oznacza mniej niespodzianek o 3 w nocy na produkcji.

5.7 Autowire – automatyczne wstrzykiwanie zależności

Autowire to serce naszego kontenera. Analizuje konstruktor klasy i automatycznie dostarcza wszystkie wymagane zależności.

5.7.1 Interfejs Autowire

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Contracts;

interface AutowireContract
{
    public function instantiate(string $className): object
        ;
}
```

Listing 5.6: Interfejs AutowireContract

5.7.2 Implementacja Autowire

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Resolvers;

use Larafony\Framework\Container\Contracts\
    AutowireContract;
```

```
8  use Larafony\Framework\Container\Contracts\
       ReflectionResolverContract;
9  use Larafony\Framework\Container\Exceptions\NotFoundError;
10 use Psr\Container\ContainerInterface;
11 use ReflectionClass;
12 use ReflectionException;
13 use ReflectionParameter;
14
15 /**
16  * Class Autowire
17  *
18  * Responsible for automatically resolving and injecting
       dependencies.
19  */
20 class Autowire implements AutowireContract
21 {
22     public function __construct(
23         private ContainerInterface $container,
24         private ?ReflectionResolverContract $resolver =
              null,
25     ) {
26         $this->resolver ??= new ReflectionResolver();
27     }
28
29     /**
30      * Autowire and instantiate a class.
31      *
32      * @template T of object
33      *
34      * @param class-string<T> $className The name of the
            class to instantiate
35      *
36      * @return T The instantiated object
37      *
38      * @throws ReflectionException If the class cannot be
            reflected
39      * @throws NotFoundError If a dependency cannot be
            resolved
40      */
41     public function instantiate(string $className): object
42     {
```

```
43          $reflectionClass = new ReflectionClass($className);
44          $constructor = $reflectionClass->getConstructor();
45
46          if ($constructor === null) {
47              return $reflectionClass->newInstance();
48          }
49
50          $parameters = $this->resolver?->
                getConstructorParameters($className) ?? [];
51          $arguments = $this->resolveParameters($parameters);
52
53          return $reflectionClass->newInstanceArgs(
                $arguments);
54      }
55
56      /**
57       * Resolve parameters for dependency injection.
58       *
59       * @param array<int, ReflectionParameter> $parameters
60       *
61       * @return array<mixed>
62       */
63      private function resolveParameters(array $parameters):
            array
64      {
65          return array_map(function (ReflectionParameter
                $parameter) {
66              $parameterName = $parameter->getName();
67              /** @var string $type */
68              $type = $this->resolver?->getParameterType(
                    $parameter);
69
70              return match (true) {
71                  // 1. return given value if exists
72                  $this->container->has($parameterName) =>
                        $this->container->get($parameterName),
73                  $this->container->has($type) => $this->
                        container->get($type),
74                  // 2. return default value if exist
```

```
75                  $this->resolver?->hasDefaultValue(
                        $parameter) => $this->resolver->
                        getDefaultValue($parameter),
76
77                  // 3. return null if allowed
78                  $this->resolver?->allowsNull($parameter)
                        => null,
79
80                  // 4. for builtin types return default
                        value
81                  $type && $this->isBuiltInType($type) =>
                        $this->resolver?->
                        getDefaultValueForBuiltInType($type),
82
83                  // 5. for object check recursively
84                  $type && class_exists($type) => $this->
                        instantiate($type),
85                  // otherwise throw not found exception
86                  default => throw new NotFoundError(
87                      "Unable to resolve parameter {
                            $parameterName} of type {$type}"
88                  )
89              };
90          }, $parameters);
91      }
92
93      private function isBuiltInType(string $type): bool
94      {
95          return in_array($type, ['int', 'float', 'string',
                'bool', 'array']);
96      }
97  }
```

Listing 5.7: Klasa Autowire

5.7.3 Algorytm rozwiązywania zależności

Metoda `resolveParameters()` sprawdza parametry w następującej kolejności:

1. **Nazwa parametru** – czy kontener ma wartość pod kluczem równym nazwie parametru?
2. **Typ parametru** – czy kontener ma wartość pod kluczem równym typowi?
3. **Wartość domyślna** – czy parametr ma zdefiniowaną wartość domyślną?
4. **Nullable** – czy parametr może być `null`?
5. **Typ wbudowany** – dla `int`, `string`, itp. zwracamy sensowne domyślne wartości
6. **Klasa** – rekursywnie tworzymy instancję wymaganej klasy
7. **Błąd** – jeśli nic nie pasuje, rzucamy `NotFoundError`

Nota autora

Użycie wyrażenia `match` z PHP 8 zamiast łańcucha `if-else` sprawia, że algorytm jest czytelny jak przepis kulinarny. No, może jak przepis dla programistów – z mniejszą ilością masła. Dodatkowo `match` ma 2 przewagi nad `switch`:

- Używa `===` zamiast `==`
- Rzuci wyjątkiem jeśli nic nie zostanie dopasowane, co pozwala łatwiej wykryć potencjalne błędy przy statycznej analizie kodu

5.8 DotContainer – storage z kropką

Konfiguracja w prawdziwych aplikacjach jest zagnieżdżona: `database.mysql.host`, `cache.redis.port`. Klasa `DotContainer` obsługuje taką notację naturalnie.

5.8.1 Interfejs ArrayContract

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Contracts;

interface ArrayContract
{
    public function get(string $key, mixed $default = null
        ): mixed;
    public function set(string $key, mixed $default = null
        ): void;
    public function has(string $key): bool;
}
```

Listing 5.8: Interfejs ArrayContract

5.8.2 Pomocnik ArrayGet

Klasa do odczytu zagnieżdżonych wartości:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Helpers;

use ArrayObject;

final readonly class ArrayGet
{
    public function __construct(
        private DotContainer $container
```

```
    ) {
    }

    public function get(string $key, mixed $default = null
        ): mixed
    {
        $keys = explode('.', $key);
        $data = $this->container->getArrayCopy();
        $data = new ArrayObject($data);

        foreach ($keys as $segment) {
            if ($data->offsetExists($segment)) {
                $value = $data[$segment];
                $value = is_array($value) ? new
                    ArrayObject($value) : $value;
                $data = $value;
            } else {
                return $default;
            }
        }

        return $data;
    }
}
```

Listing 5.9: Klasa ArrayGet

5.8.3 Pomocnik ArraySet

Klasa do zapisu zagnieżdżonych wartości z użyciem referencji:

```
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Helpers;

```

```php
use ArrayObject;

final class ArraySet
{
    public function __construct(
        private DotContainer $container
    ) {
    }

    public function set(string $key, mixed $value): void
    {
        $keys = explode('.', $key);
        $data = &$this->container;

        foreach ($keys as $segment) {
            if (! isset($data[$segment])) {
                $data[$segment] = new ArrayObject();
            }

            $data = &$data[$segment];
        }

        $data = $value;
    }
}
```

Listing 5.10: Klasa ArraySet

5.8.4 DotContainer

Połączenie powyższych helperów w jedną klasę:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Helpers;
```

```php

use ArrayObject;
use Larafony\Framework\Container\Contracts\ArrayContract;

/**
 * @extends ArrayObject<string|int, mixed>
 */
class DotContainer extends ArrayObject implements
   ArrayContract
{
    private readonly ArrayGet $arrayGet;
    private readonly ArraySet $arraySet;

    /**
     * @param array<int|string, mixed> $array
     */
    public function __construct(array $array = [])
    {
        parent::__construct($array);
        $this->arrayGet = new ArrayGet($this);
        $this->arraySet = new ArraySet($this);
    }

    public function get(string $key, mixed $default = null
       ): mixed
    {
        return $this->arrayGet->get($key, $default);
    }

    public function set(string $key, mixed $default = null
       ): void
    {
        $this->arraySet->set($key, $default);
    }

    public function has(string $key): bool
    {
        return $this->arrayGet->get($key) !== null;
    }
}
```

Listing 5.11: Klasa DotContainer

Nota autora

Dot notation to jeden z tych wzorców, który gdy już poznasz, nie wyobrażasz sobie życia bez niego. `$config->get('database.mysql.host')` jest po prostu czytelniejsze niż `$config['database']['mysql']['host']` – i nie wymaga sprawdzania, czy każdy poziom istnieje.

5.9 Główna klasa Container

Czas połączyć wszystkie elementy w jedną całość:

5.9.1 Atrybut #[NoDiscard] – nowość PHP 8.5

W klasie `Container` przy metodach `get()` i `getBinding()` znajdziesz atrybut `#[NoDiscard]`. To nowość wprowadzona w PHP 8.5, która informuje analizatory statyczne (PHPStan, Psalm) oraz IDE, że wartość zwracana przez metodę **nie powinna być ignorowana**.

```
#[NoDiscard]
public function get(string $id): mixed { ... }

// Poprawne użycie:
$service = $container->get(UserService::class);

// Błąd - wynik jest ignorowany:
$container->get(UserService::class);  // PHPStan zgłosi ostrzeżenie
```

Dlaczego to ważne? Wywołanie `get()` bez przypisania wyniku to prawie na pewno błąd programisty – albo zapomniał przypisać wartość, albo pomylił metodę. Dzięki `#[NoDiscard]` taki błąd zostanie wykryty podczas analizy

statycznej, a nie dopiero na produkcji.

Nota autora

Atrybut `#[NoDiscard]` to jeden z tych dodatków, które sprawiają, że PHP staje się coraz bardziej "dorosłym" językiem. Rust ma podobny mechanizm od lat i programiści go uwielbiają. Teraz PHP dołącza do klubu – lepiej późno niż wcale.

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container;

use Larafony\Framework\Container\Contracts\ArrayContract;
use Larafony\Framework\Container\Contracts\
   AutowireContract;
use Larafony\Framework\Container\Contracts\
   ContainerContract;
use Larafony\Framework\Container\Exceptions\NotFoundError;
use Larafony\Framework\Container\Helpers\DotContainer;
use Larafony\Framework\Container\Resolvers\Autowire;
use Larafony\Framework\Core\Support\Str;
use NoDiscard;

class Container implements ContainerContract
{
    /**
     * @var array<string, string|int|float|bool|null>
     */
    private array $bindings = [];

    public function __construct(
        private ?AutowireContract $autowire = null,
        private readonly ArrayContract $entries = new
           DotContainer(),
    ) {
        $this->autowire ??= new Autowire($this);
    }
```

```
29      public function bind(string $key, float|bool|int|
            string|null $value): void
30      {
31          $this->bindings[$key] = $value;
32      }
33
34      #[NoDiscard]
35      public function getBinding(string $key): string|int|
            float|bool|null
36      {
37          return $this->bindings[$key] ?? throw new
                NotFoundError('Binding not found: '.$key);
38      }
39
40      /**
41       * Sets item in container
42       *
43       * @param string $key
44       * @param mixed $value
45       *
46       * @return self
47       */
48      public function set(string $key, mixed $value):
            Contracts\ContainerContract
49      {
50          if (Str::isClassString($value) && ! $this->has(
                $value)) {
51              $value = $this->autowire?->instantiate($value)
                    ;
52          }
53          $this->entries->set($key, $value);
54          return $this;
55      }
56
57      #[NoDiscard]
58      public function get(string $id): mixed
59      {
60          if (! $this->entries->has($id)) {
61              return $this->autowire?->instantiate($id);
62          }
63
```

```
64          $value = $this->entries->get($id);
65
66          return Str::isClassString($value) ? $this->
                autowire?->instantiate($value) : $value;
67      }
68
69      public function has(string $id): bool
70      {
71          return $this->entries->has($id);
72      }
73  }
```

Listing 5.12: Klasa Container

5.9.2 Dwa systemy przechowywania

Kontener używa dwóch oddzielnych systemów:

- `$bindings` – dla prostych wartości skalarnych (konfiguracja)
- `$entries` (DotContainer) – dla obiektów i złożonych struktur

Rozdzielenie to pozwala na szybki dostęp do konfiguracji bez przeszukiwania całego kontenera.

5.9.3 Lazy autowiring

W metodzie `get()` kontener sprawdza, czy żądany klucz istnieje. Jeśli nie – próbuje utworzyć instancję klasy o tej nazwie. To oznacza, że nie musisz rejestrować każdej klasy – kontener sam ją znajdzie i utworzy.

Nota autora

To podejście ma nazwę "convention over configuration" – jeśli klasa nazywa się `UserRepository`, kontener założy, że chcesz właśnie ją. Mniej konfiguracji, więcej pisania kodu, który robi rzeczy.

5.10 Service Provider – modularność usług

W dużych aplikacjach rejestrowanie wszystkich usług w jednym miejscu szybko staje się nieczytelne. Service Provider pozwala grupować powiązane usługi w oddzielne moduły.

5.10.1 Interfejs ServiceProvider

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Container\Contracts;

interface ServiceProviderContract
{
    /**
     * @return array<int|string, class-string>
     */
    public function providers(): array;
    /**
     * Register services in the given container.
     *
     * @param ContainerContract $container The container
        to register services in
     */
    public function register(ContainerContract $container)
        : self;

```

```
20      /**
21       * bootstrapping services
22       */
23      public function boot(ContainerContract $container):
            void;
24  }
```

Listing 5.13: Interfejs ServiceProviderContract

5.10.2 Implementacja ServiceProvider

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Container;
6
7   use Larafony\Framework\Container\Contracts\
        ContainerContract;
8   use Larafony\Framework\Container\Contracts\
        ServiceProviderContract;
9
10  abstract class ServiceProvider implements
        ServiceProviderContract
11  {
12      /**
13       * @return array<int|string, class-string>
14       */
15      public function providers(): array
16      {
17          return [];
18      }
19
20      public function register(ContainerContract $container)
            : self
21      {
22          foreach ($this->providers() as $key => $class) {
```

```
23              $key = is_int($key) ? $class : $key;
24              $container->set($key, $class);
25          }
26          return $this;
27      }
28
29      public function boot(ContainerContract $container):
            void
30      {
31          if (! $container->has($this::class)) {
32              $container->set($this::class, $this);
33          }
34      }
35  }
```

Listing 5.14: Klasa ServiceProvider

5.10.3 Cykl życia Service Providera

1. **providers()** – zwraca tablicę klas do zarejestrowania
2. **register()** – rejestruje wszystkie klasy w kontenerze
3. **boot()** – wykonuje dodatkową inicjalizację po rejestracji

Metoda `register()` inteligentnie obsługuje klucze tablicy:

- Klucz liczbowy: klasa rejestrowana pod własną nazwą
- Klucz tekstowy: klasa rejestrowana pod podanym aliasem (np `[Logger-Contract => MyLogger]`)

Nota autora

Ten wzorzec jest inspirowany Laravelem, ale uproszczony. W Larafony `boot()` służy głównie do samoregistracji providera w kontenerze – przydatne, gdy chcesz później sprawdzić, które providery zostały załadowane.

Nota autora

Jedynymi zależnościami produkcyjnymi Larafony są pakiety PSR. Oznacza to, że jeśli kiedykolwiek zechcesz podmienić któryś z komponentów frameworka – wystarczy utworzyć ServiceProvider, w którym do odpowiedniego klucza (interfejsu) przypiszesz własną implementację. Tylko i aż tyle. Żadnego vendor lock-in, żadnych haków w kodzie frameworka. Twoja aplikacja, Twoje zasady.

5.11 Testy jednostkowe

5.11.1 Klasy pomocnicze do testów

Do testowania kontenera potrzebujemy kilku prostych klas:

```php
class Database {}

class UserRepository {
    public function __construct(
        public readonly Database $database,
    ) {}
}

class TestClassWithParams {
    public function __construct(
        public string $name,
        public int $age,
    ) {}
}

class TestClassWithDefaults {
    public function __construct(
        public string $required,
        public string $optional = 'default',
    ) {}
}
```

```
22
23 class TestClassWithNullable {
24     public function __construct(
25         public ?string $nullable,
26         public string $required,
27     ) {}
28 }
```

5.11.2 Testy ReflectionResolver

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Larafony\Container;
6
7  use Larafony\Framework\Container\Exceptions\ContainerError
       ;
8  use Larafony\Framework\Container\Resolvers\
       ReflectionResolver;
9  use PHPUnit\Framework\TestCase;
10 use ReflectionParameter;
11
12 final class ReflectionResolverTest extends TestCase
13 {
14     private ReflectionResolver $resolver;
15
16     protected function setUp(): void
17     {
18         $this->resolver = new ReflectionResolver();
19     }
20
21     public function testGetConstructorParameters(): void
22     {
23         $parameters = $this->resolver->
               getConstructorParameters(TestClassWithParams::
```

```
            class);

        $this->assertCount(2, $parameters);
        $this->assertInstanceOf(ReflectionParameter::class
            , $parameters[0]);
    }

    public function testGetParameterType(): void
    {
        $parameters = $this->resolver->
            getConstructorParameters(TestClassWithParams::
            class);

        $type = $this->resolver->getParameterType(
            $parameters[0]);
        $this->assertSame('string', $type);
    }

    public function testHasDefaultValue(): void
    {
        $parameters = $this->resolver->
            getConstructorParameters(TestClassWithDefaults
            ::class);

        $this->assertFalse($this->resolver->
            hasDefaultValue($parameters[0]));
        $this->assertTrue($this->resolver->hasDefaultValue
            ($parameters[1]));
    }

    public function testGetDefaultValue(): void
    {
        $parameters = $this->resolver->
            getConstructorParameters(TestClassWithDefaults
            ::class);

        $this->assertSame('default', $this->resolver->
            getDefaultValue($parameters[1]));
    }

```

```
52      public function testGetDefaultValueThrowsException():
            void
53      {
54          $parameters = $this->resolver->
                getConstructorParameters(TestClassWithDefaults
                ::class);
55
56          $this->expectException(ContainerError::class);
57          $this->expectExceptionMessage('has no default
                value');
58
59          $this->resolver->getDefaultValue($parameters[0]);
60      }
61
62      public function testAllowsNull(): void
63      {
64          $parameters = $this->resolver->
                getConstructorParameters(TestClassWithNullable
                ::class);
65
66          $this->assertTrue($this->resolver->allowsNull(
                $parameters[0]));
67          $this->assertFalse($this->resolver->allowsNull(
                $parameters[1]));
68      }
69
70      public function testGetDefaultValueForBuiltInType():
            void
71      {
72          $this->assertSame(0, $this->resolver->
                getDefaultValueForBuiltInType('int'));
73          $this->assertSame(0.0, $this->resolver->
                getDefaultValueForBuiltInType('float'));
74          $this->assertSame('', $this->resolver->
                getDefaultValueForBuiltInType('string'));
75          $this->assertSame(false, $this->resolver->
                getDefaultValueForBuiltInType('bool'));
76          $this->assertSame([], $this->resolver->
                getDefaultValueForBuiltInType('array'));
77          $this->assertNull($this->resolver->
                getDefaultValueForBuiltInType('unknown'));
```

```
78      }
79  }
```

Listing 5.15: Testy klasy ReflectionResolver

5.11.3 Testy DotContainer

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Tests\Container;
6
7   use ArrayObject;
8   use Larafony\Framework\Container\Helpers\DotContainer;
9   use PHPUnit\Framework\TestCase;
10
11  final class DotContainerTest extends TestCase
12  {
13      public function testSetAndGetWithDotNotation(): void
14      {
15          $container = new DotContainer();
16
17          $container->set('foo.bar.baz', 'value');
18          $this->assertEquals('value', $container->get('foo.
                bar.baz'));
19
20          $container->set('foo.bar.qux', 123);
21          $this->assertEquals(123, $container->get('foo.bar.
                qux'));
22      }
23
24      public function testSetArray(): void
25      {
26          $container = new DotContainer();
27          $container->set('test', ['foo' => 'bar', 'baz' =>
                'qux']);
```

```
28          $this->assertEquals('bar', $container->get('test.
               foo'));
29          $this->assertEquals('qux', $container->get('test.
               baz'));
30      }
31
32      public function testGetWithDefaultValue(): void
33      {
34          $container = new DotContainer();
35
36          $this->assertEquals('default', $container->get('
               non.existent.key', 'default'));
37      }
38
39      public function testNestedArrayObjectIsCreated(): void
40      {
41          $container = new DotContainer();
42
43          $container->set('foo.bar.baz', 'value');
44          $this->assertInstanceOf(ArrayObject::class,
               $container['foo']);
45          $this->assertInstanceOf(ArrayObject::class,
               $container['foo']['bar']);
46      }
47  }
```

Listing 5.16: Testy klasy DotContainer

5.11.4 Testy Container

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Tests\Container;
6
7   use Larafony\Framework\Container\Container;
```

```php
use Larafony\Framework\Web\Application;
use PHPUnit\Framework\TestCase;

final class ContainerTest extends TestCase
{
    private Container $container;

    protected function setUp(): void
    {
        $this->container = Application::instance();
    }

    public function testBindings(): void
    {
        $this->container->bind('const', 1);
        $this->assertEquals(1, $this->container->
            getBinding('const'));
    }

    public function testAutowireSimpleClass(): void
    {
        $instance = $this->container->get(Database::class)
            ;
        $this->assertInstanceOf(Database::class, $instance
            );
    }

    public function testAutowireWithDependencies(): void
    {
        $instance = $this->container->get(UserRepository::
            class);
        $this->assertInstanceOf(UserRepository::class,
            $instance);
        $this->assertInstanceOf(Database::class, $instance
            ->database);
    }

    public function testSetAndGet(): void
    {
        $this->container->set('key', 'value');
        $this->assertTrue($this->container->has('key'));
```

```
43          $this->assertEquals('value', $this->container->get
                ('key'));
44      }
45
46      public function testHasReturnsFalseForNonExistent():
            void
47      {
48          $this->assertFalse($this->container->has('
                nonexistent'));
49      }
50
51      public function testRegisterByClassName()
52      {
53          $this->container->set(Database::class, Database::
                class);
54          $db = $this->container->get(Database::class);
55          $this->assertInstanceOf(Database::class, $db);
56      }
57  }
```

Listing 5.17: Testy klasy Container

5.12 Podsumowanie

W tym rozdziale zbudowaliśmy kompletny kontener Dependency Injection zgodny z PSR-11:

- Interfejs `ContainerContract` rozszerzający PSR-11 o praktyczne metody
- Wyjątki `ContainerError` i `NotFoundError` zgodne ze standardem
- `ReflectionResolver` jako wrapper na PHP Reflection API
- `Autowire` automatycznie rozwiązujący zależności konstruktora
- `DotContainer` z obsługą zagnieżdżonej konfiguracji

- `ServiceProvider` do modularnej organizacji usług
- Lazy autowiring – klasy tworzone na żądanie bez jawnej rejestracji

Nota autora

Nasz kontener jest celowo uproszczony względem kombajnów jak Symfony Container czy PHP-DI. Nie ma dekoratorów, factory definitions ani tagged services. Ale ma wszystko, czego potrzebujemy do zbudowania działającego frameworka – a to jest celem tej książki.
W kolejnych rozdziałach wykorzystamy kontener do rejestracji routera, middleware'ów i innych komponentów HTTP. Zobaczysz, jak DI zmienia chaotyczny kod w elegancką orkiestrację obiektów.
Czas na routing!

Rozdział 10

DBAL – schemat bazy danych

W nadchodzących czterech rozdziałach będziemy wdrażać pełny, choć uproszczony, mechanizm zarządzania bazą danych, znany jako Database Abstraction Layer (DBAL). Naszym celem będzie stworzenie elastycznego systemu, który pozwoli na łatwą interakcję z bazą danych, zarówno w kontekście zarządzania strukturą bazy, jak i operacji na danych.

W pierwszej części (Rozdział 10) skupimy się na podstawowym mechanizmie połączenia z bazą danych oraz na systemie `Grammar` – wzorcu inspirowanym Laravelem, który pozwala na eleganckie generowanie zapytań SQL z definicji obiektowych. Zbudujemy fluent API do definiowania tabel, które będzie czytelne i intuicyjne w użyciu.

W Rozdziale 11 przejdziemy do tworzenia klasy `QueryBuilder`. `QueryBuilder` zaimplementuje większość operacji związanych z danymi w bazie MySQL: wstawianie danych (`INSERT`), ich wybieranie (`SELECT`), aktualizacje (`UPDATE`) oraz usuwanie (`DELETE`).

W Rozdziale 12 dodamy pierwsze komendy do zarządzania schematami bazy danych, niezbędne do pracy z migracjami. Stworzymy komendę `make:migration` oraz system szablonów, a następnie zaimplementujemy komendę `migrate`.

Zwieńczeniem naszych prac będzie Rozdział 13, w którym wdrożymy pełny ORM z obsługą relacji oraz komendy do generowania modeli, fabryk i seederów.

Nota autora

Poniżej przedstawiono przykładowy proces automatycznego seedowania bazy danych przy wykorzystaniu pełnego potencjału PHP 8.5. Stworzenie migracji dla tabeli użytkowników:

```
1 php console/bin make:migration --table=users
```

Następnie musimy uzupełnić tę migrację zgodnie z zaplanowaną strukturą bazy danych.

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DummyRootNamespace;
6
7  use Larafony\Framework\Database\Base\Migrations\Migration;
8  use Larafony\Framework\Database\Schema;
9
10 return new class extends Migration
11 {
12     /**
13      * run migration.
14      */
15     public function up(): void
16     {
17         Schema::create('users', function ($table) {
18             $table->id();
19             $table->string('email', 100);
20             $table->string('username', 50)->nullable(true)
                   ;
21             $table->string('password');
22             $table->string('remember_token', 100)->
                   nullable(true);
23             $table->string('password_reset_token', 100)->
                   nullable(true);
```

```
24              $table->timestamp('password_reset_expires')->
                    nullable(true);
25              $table->timestamp('email_verified_at')->
                    nullable(true);
26              $table->integer('is_active', 'TINYINT')->
                    length(1)->default(1);
27              $table->timestamp('last_login_at')->nullable(
                    true);
28              $table->timestamps();
29
30              // Indexes
31              $table->unique('email');
32              $table->unique('username');
33              $table->index('is_active');
34          }) |> Schema::execute(...);
35      }
36
37      /**
38       * rollback migration.
39       */
40      public function down(): void
41      {
42          Schema::dropIfExists('users') |> Schema::execute
                (...);
43      }
44  };
```

Listing 10.1: Migracja tabeli users

Nota autora

Powyższy kod to szablon, który wykorzystamy w rozdziale 18, gdy demo-app zostanie wydzielone jako osobny projekt i stworzymy pełne komendy do zarządzania migracjami. Na razie, jeśli chcesz użyć tej klasy, po prostu zamień wszystkie placeholdery na właściwe wartości.

Drugim krokiem jest w pełni **automatyczne** wygenerowanie modelu. Jedyne co musimy zrobić ręcznie to ustawić atrybuty zasilające naszą fabrykę danych.

```
php console/bin make:model User --table=users
```

```
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\ORM\Entities;

use Larafony\Framework\Cache\Cache;
use Larafony\Framework\Clock\ClockFactory;
use Larafony\Framework\Clock\Contracts\Clock;
use Larafony\Framework\Database\ORM\Attributes\
   BelongsToMany;
use Larafony\Framework\Database\ORM\Attributes\CastUsing;
use Larafony\Framework\Database\ORM\Model;

class User extends Model
{
    public string $email {
        get => $this->email;
        set {
            $this->email = $value;
            $this->markPropertyAsChanged('email');
        }
    }
    public string $username {
        get => $this->username;
        set {
            $this->username = $value;
            $this->markPropertyAsChanged('username');
        }
    }
    public string $password {
        get => $this->password;
        set {
            // Auto-hash password if not already hashed (
               Argon2id)
```

```
34              $this->password = str_starts_with($value, '
                    $argon2') ? $value : password_hash($value,
                    PASSWORD_ARGON2ID);
35              $this->markPropertyAsChanged('password');
36          }
37      }
38      public ?string $remember_token {
39          get => $this->remember_token;
40          set {
41              $this->remember_token = $value;
42              $this->markPropertyAsChanged('remember_token')
                    ;
43          }
44      }
45      public ?string $password_reset_token {
46          get => $this->password_reset_token;
47          set {
48              $this->password_reset_token = $value;
49              $this->markPropertyAsChanged('
                    password_reset_token');
50          }
51      }
52      //this is just for now, in final version it will
            attribute based casting to Clock instance
53      public ?string $password_reset_expires {
54          get => $this->password_reset_expires;
55          set {
56              $this->password_reset_expires = $value;
57              $this->markPropertyAsChanged('
                    password_reset_expires');
58          }
59      }
60      public ?Clock $email_verified_at {
61          get => $this->email_verified_at;
62          set {
63              $this->email_verified_at = $value;
64              $this->markPropertyAsChanged('
                    email_verified_at');
65          }
66      }
67      public int $is_active {
```

```
68          get => $this->is_active;
69          set {
70              $this->is_active = $value;
71              $this->markPropertyAsChanged('is_active');
72          }
73      }
74      //enough for now, other properties will be added after
            full support of ORM is done
75      //continue reading ...
76  }
```

Listing 10.2: Model User

Nota autora

Użyty w kodzie powyżej atrybut `FakeAs` przyjmuje jako parametr enum ze zdefiniowanymi metodami dla klasy `Faker/Generator`. Natomiast sam model w pełni czerpie z wprowadzonego w PHP 8.4 mechanizmu Property Hooks pozwalającego na śledzenie prawdziwych pól bez użycia metod magicznych `__get` i `__set`.

Nota autora

Pełne uruchomienie tej klasy będzie możliwe dopiero po skończeniu rozdziału 13 gdzie k

Ostatnim krokiem jest stworzenie seedera i wypełnienie bazy danych:

```
1  php console/bin make:seeder UserSeeder
2  php console/bin database:seed
```

```
1  <?php
2
3  declare(strict_types=1);
4
```

```php
namespace App\Database\Seeders;

use App\Models\User;
use Larafony\Framework\Database\ORM\Entities\Permission;
use Larafony\Framework\Database\ORM\Entities\Role;

class UserSeeder
{
    public function run(): void
    {
        // Create admin user
        $admin = new User();
        $admin->email = 'admin@example.com';
        $admin->username = 'admin';
        $admin->password = 'password'; // Auto-hashed with
            Argon2id
        $admin->is_active = 1;
        $admin->save();

        // Create regular user
        $user = new User();
        $user->email = 'user@example.com';
        $user->username = 'user';
        $user->password = 'password'; // Auto-hashed with
           Argon2id
        $user->is_active = 1;
        $user->save();

        // Create roles
        $adminRole = new Role();
        $adminRole->name = 'admin';
        $adminRole->description = 'Administrator role';
        $adminRole->save();

        $userRole = new Role();
        $userRole->name = 'user';
        $userRole->description = 'Regular user role';
        $userRole->save();

        // Create permissions
        $addNotePermission = new Permission();
```

```
44          $addNotePermission->name = 'notes.create';
45          $addNotePermission->description = 'Can create
               notes';
46          $addNotePermission->save();
47
48          $editNotePermission = new Permission();
49          $editNotePermission->name = 'notes.edit';
50          $editNotePermission->description = 'Can edit notes
               ';
51          $editNotePermission->save();
52
53          $deleteNotePermission = new Permission();
54          $deleteNotePermission->name = 'notes.delete';
55          $deleteNotePermission->description = 'Can delete
               notes';
56          $deleteNotePermission->save();
57
58          // Assign roles to users
59          $admin->addRole($adminRole);
60          $user->addRole($userRole);
61      }
62  }
```

Listing 10.3: Seeder UserSeeder

10.1 Wprowadzenie do DBAL

Baza danych stanowi serce każdej aplikacji, w której przechowywanie i zarządzanie danymi ma kluczowe znaczenie. W naszym frameworku podejście do obsługi bazy danych oparte jest na DBAL – warstwie abstrakcji nad bazą danych, pozwalającej na eleganckie i przejrzyste zarządzanie schematem oraz operacjami na danych.

DBAL to koncepcja pozwalająca na abstrakcyjne podejście do operacji bazodanowych, odczytując je jako zestaw obiektów i operacji, a nie jako surowe zapytania SQL. Naszym celem jest stworzenie rozwiązania, które nie tylko ułatwi operacje na bazie danych, ale także pozwoli na łatwą rozbudowę i obsługę

różnych silników bazodanowych.

Nota autora

Architektura naszego DBAL jest inspirowana Laravelem i opiera się na wzorcu `Grammar`. Dzięki temu możemy łatwo dodawać wsparcie dla różnych baz danych – wystarczy stworzyć nową klasę `Grammar` dla danego silnika (np. PostgreSQL, SQLite), bez modyfikacji reszty kodu.

10.2 Architektura DBAL

Nasz system DBAL składa się z następujących komponentów:

- `ConnectionContract` – interfejs definiujący połączenie z bazą danych
- `GrammarContract` – interfejs kompilatora SQL dla danego silnika
- `SchemaBuilder` – główna klasa do zarządzania schematem
- `TableDefinition` – fluent API do definiowania struktury tabeli
- `BaseColumn` i klasy pochodne – definicje kolumn z metodami fluent
- `IndexDefinition` i klasy pochodne – definicje indeksów

Struktura katalogów odzwierciedla tę architekturę:

```
Database/
+-- Base/                    # Abstrakcyjne klasy bazowe
|   +-- Contracts/           # Interfejsy
|   +-- Schema/              # Definicje schematow
|   |   +-- Columns/         # Bazowe klasy kolumn
|   |   +-- IndexDefinitions/
|   |   +-- Builders/
|   +-- Query/               # QueryBuilder (rozdzial 11)
+-- Drivers/
    +-- MySQL/               # Implementacja dla MySQL
```

```
        +-- Schema/
        |   +-- ColumnDefinitions/
        |   +-- IndexDefinitions/
        |   +-- Builders/
        +-- Query/
```

10.3 Interfejs ConnectionContract

Podstawą każdej warstwy komunikacji z bazą danych jest połączenie. Interfejs `ConnectionContract` definiuje metody, które każda implementacja musi obsługiwać:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Base\Contracts;

interface ConnectionContract
{
    public function connect(): void;

    /**
     * @param array<string|int, string|float|int|null>
        $params
     */
    public function query(string $sql, array $params = [])
       : \PDOStatement;

    /**
     * @return array<int, int|false>
     */
    public function getConnectionOptions(): array;

    public function getLastInsertId(): ?string;
```

```php

    public function quote(int|float|string|bool|null $value): string;

    /**
     * Execute a SELECT query and return all results
     * Cursor is automatically closed after fetching
     *
     * @param string $sql
     * @param array<int, mixed> $params
     *
     * @return array<int, array<string, mixed>>
     */
    public function select(string $sql, array $params = []): array;

    /**
     * Execute an INSERT/UPDATE/DELETE query and return affected rows
     * Cursor is automatically closed after getting row count
     *
     * @param string $sql
     * @param array<int, mixed> $params
     *
     * @return int Number of affected rows
     */
    public function execute(string $sql, array $params = []): int;
}
```

Listing 10.4: Interfejs ConnectionContract

Nota autora

Zwróć uwagę na metody `select()` i `execute()`. To uproszczenie typowych operacji: `select()` automatycznie zamyka kursor po pobraniu wyników, `execute()` zwraca liczbę zmienionych wierszy. Dzięki temu unikamy wycieków pamięci związanych z otwartymi kursorami.

10.4 Wzorzec Grammar

Serce naszego DBAL to wzorzec `Grammar` – klasa odpowiedzialna za kompilację obiektowych definicji tabel do surowego SQL. Dzięki temu rozdzieleniu logika definiowania schematu jest niezależna od składni konkretnego silnika bazy danych.

Nota autora

Wzorzec Grammar to jedna z najlepszych decyzji architektonicznych w Laravelu. Zamiast generować SQL bezpośrednio w `SchemaBuilder`, delegujemy to do osobnej klasy. Korzyści:

- **Separacja odpowiedzialności**: `SchemaBuilder` wie *co* robić (twórz tabelę, modyfikuj kolumnę), `Grammar` wie *jak* to zapisać w SQL
- **Łatwe testowanie**: `Grammar` to czysta funkcja – wchodzi `TableDefinition`, wychodzi string SQL. Bez mocków, bez bazy danych
- **Wsparcie wielu baz**: Nowa baza = nowa klasa `Grammar`. Zero zmian w istniejącym kodzie

W praktyce oznacza to, że Twoje migracje napisane dla MySQL zadziałają na PostgreSQL po zmianie jednej linijki konfiguracji – zakładając, że zaimplementujesz `PostgreSQL/Grammar`.

10.4.1 Interfejs GrammarContract

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Base\Contracts;

use Larafony\Framework\Database\Base\Schema\
    TableDefinition;
```

```php

interface GrammarContract
{
    public function compileCreate(TableDefinition $table): string;
    public function compileAddColumns(TableDefinition $table): string;
    public function compileModifyColumns(TableDefinition $table): string;
    public function compileDropColumns(TableDefinition $table): string;
    public function compileDropTable(string $table, bool $ifExists = false): string;
}
```

Listing 10.5: Interfejs GrammarContract

10.4.2 Implementacja MySQL Grammar

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL;

use Larafony\Framework\Database\Base\Contracts\GrammarContract;
use Larafony\Framework\Database\Base\Schema\TableDefinition;
use Larafony\Framework\Database\Drivers\MySQL\Schema\Builders\AddColumns;
use Larafony\Framework\Database\Drivers\MySQL\Schema\Builders\ChangeColumns;
use Larafony\Framework\Database\Drivers\MySQL\Schema\Builders\CreateTableBuilder;
use Larafony\Framework\Database\Drivers\MySQL\Schema\Builders\DropColumns;
```

```
13
14 class Grammar implements GrammarContract
15 {
16     public function compileCreate(TableDefinition $table):
           string
17     {
18         return new CreateTableBuilder()->build($table);
19     }
20
21     public function compileAddColumns(TableDefinition
           $table): string
22     {
23         return new AddColumns()->build($table);
24     }
25
26     public function compileModifyColumns(TableDefinition
           $table): string
27     {
28         return new ChangeColumns()->build($table);
29     }
30
31     public function compileDropColumns(TableDefinition
           $table): string
32     {
33         return new DropColumns()->build($table);
34     }
35
36     public function compileDropTable(string $table, bool
           $ifExists = false): string
37     {
38         return sprintf('DROP TABLE %s%s', $ifExists ? 'IF
               EXISTS ' : '', $table);
39     }
40 }
```

Listing 10.6: Klasa MySQL Grammar

Nota autora

Klasa `Grammar` deleguje kompilację do wyspecjalizowanych builderów. To kolejny poziom rozdzielenia odpowiedzialności – `Grammar` wie *co* kompilować, buildery wiedzą *jak*. Dzięki temu dodanie nowego typu operacji (np. `RENAME TABLE`) wymaga tylko nowego buildera.

10.5 TableDefinition – fluent API

Klasa `TableDefinition` to serce naszego fluent API do definiowania tabel. Pozwala na czytelne i intuicyjne definiowanie struktury tabeli:

```
$schema->create('users', function (TableDefinition $table) {
    $table->id();
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamps();
});
```

10.5.1 MySQL TableDefinition

```
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema;

use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\DateTimeColumn;
```

```
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\EnumColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\IntColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\StringColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\TextColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    IndexDefinitions\NormalIndex;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    IndexDefinitions\PrimaryIndex;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    IndexDefinitions\UniqueIndex;

class TableDefinition extends \Larafony\Framework\Database
    \Base\Schema\TableDefinition
{
    public function integer(string $column, string $type =
        'INT'): IntColumn
    {
        $col = new IntColumn($column, type: $type);
        $this->addColumn($col);
        return $col;
    }

    public function bigInteger(string $column, string
       $type = 'MEDIUMINT'): IntColumn
    {
        return $this->integer($column, $type)->length(20);
    }
    public function smallInteger(string $column, string
       $type = 'MEDIUMINT'): IntColumn
    {
        return $this->integer($column, $type)->length(6);
    }
    public function string(string $column, int $length =
       255, string $type = 'VARCHAR'): StringColumn
    {
        $col = new StringColumn($column, length: $length,
           type: $type);
```

```
36          $this->addColumn($col);
37          return $col;
38      }
39      public function char(string $column, int $length =
            255, string $type = 'CHAR'): StringColumn
40      {
41          return $this->string($column, $length, $type);
42      }
43      public function text(string $column, string $type = '
            TEXT'): TextColumn
44      {
45          $col = new TextColumn($column, type: $type);
46          $this->addColumn($col);
47          return $col;
48      }
49      public function mediumText(string $column, string
            $type = 'MEDIUMTEXT'): TextColumn
50      {
51          return $this->text($column, $type);
52      }
53      public function longText(string $column, string $type
            = 'LONGTEXT'): TextColumn
54      {
55          return $this->text($column, $type);
56      }
57      public function json(string $column, string $type = '
            JSON'): TextColumn
58      {
59          return $this->text($column, $type);
60      }
61      public function dateTime(string $column, string $type
            = 'DATETIME'): DateTimeColumn
62      {
63          $col = new DateTimeColumn($column, type: $type);
64          $this->addColumn($col);
65          return $col;
66      }
67      public function timestamp(string $column, string $type
            = 'TIMESTAMP'): DateTimeColumn
68      {
69          return $this->dateTime($column, $type);
```

```
70      }
71      public function time(string $column, string $type = '
           TIME'): DateTimeColumn
72      {
73          return $this->dateTime($column, $type);
74      }
75      public function date(string $column, string $type = '
           DATE'): DateTimeColumn
76      {
77          return $this->dateTime($column, $type);
78      }
79
80      /**
81       * @param string|array<int, string> $columns
82       */
83      public function index(string|array $columns, ?string
           $indexName = null): NormalIndex
84      {
85          $index = new NormalIndex($this->tableName,
               $columns, $indexName);
86          $this->addIndex($index);
87          return $index;
88      }
89      /**
90       * @param string|array<int, string> $columns
91       */
92      public function primary(string|array $columns, ?string
           $indexName = null): PrimaryIndex
93      {
94          $index = new PrimaryIndex($this->tableName,
               $columns, $indexName, type: 'primary');
95          $this->addIndex($index);
96          return $index;
97      }
98      /**
99       * @param string|array<int, string> $columns
100      */
101     public function unique(string|array $columns, ?string
           $indexName = null): UniqueIndex
102     {
```

```
103         $index = new UniqueIndex($this->tableName,
                $columns, $indexName, type: 'unique');
104         $this->addIndex($index);
105         return $index;
106     }
107
108     /**
109      * @param array<int, string> $values
110      */
111     public function enum(string $column, array $values):
            EnumColumn
112     {
113         $col = new EnumColumn($column, $values);
114         $this->addColumn($col);
115         return $col;
116     }
117 }
```

Listing 10.7: Klasa MySQL TableDefinition

Nota autora

Bazowa klasa `TableDefinition` nie została tu przytoczona, znajdziesz ją w repozytorium. Jeśli chcesz, spróbuj brakujące metody (PHPStorm powinien je wykryć), zaimplementować sam - powodzenia!

Nota autora

Metody takie jak `id()`, `timestamps()`, `softDeletes()` to skróty dla często używanych wzorców. `id()` tworzy kolumnę `INT AUTO_INCREMENT` z kluczem głównym. `timestamps()` dodaje `created_at` i `updated_at`. To podejście znane z Laravela, które znacząco przyspiesza definiowanie migracji.

10.6 Klasy kolumn

Każdy typ danych MySQL jest reprezentowany przez dedykowaną klasę dziedziczącą po `BaseColumn`. Klasy te oferują fluent API do konfiguracji kolumny.

Nota autora

PHP 8.5 wprowadza nową składnię do klonowania obiektów: `clone($obj, ['field' => value])`. Pozwala ona na tworzenie niemutowalnych (immutable) obiektów z fluent API – każda metoda zwraca nową instancję zamiast modyfikować istniejącą.
Jednak w przypadku klas definiujących typy kolumn świadomie wybieramy mutowalność. Dlaczego? Bo niemutowalne podejście wymagałoby, żeby `TableDefinition` śledziło każdą zwróconą instancję:

```
// Immutable - skomplikowane, łatwo o błąd
$col = $table->integer('age');
$col = $col->unsigned(true);  // nowa instancja!
$col = $col->nullable(false); // kolejna nowa instancja!
// $table nadal ma starą referencję... plus potencjalne wycieki pamięci

// Mutable - proste i intuicyjne
$table->integer('age')->unsigned(true)->nullable(false);
// $table automatycznie ma zaktualizowaną kolumnę
```

W kontekście definiowania schematu bazy danych mutowalność jest pragmatycznym wyborem – obiekt `TableDefinition` żyje tylko przez czas wykonania callbacka, więc ryzyko niechcianych efektów ubocznych jest minimalne.

10.6.1 Bazowa klasa BaseColumn

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Base\Schema\Columns;

abstract class BaseColumn
{
    public protected(set) bool $modified = false;
    public protected(set) bool $deleted = false;
    public protected(set) bool $existsInDatabase = false;

    public function __construct(
        public readonly string $name,
        public readonly string $type,
        public bool $nullable = true,
        protected mixed $default = null,
    ) {
    }

    public function markAsExisting(): static
    {
        $this->existsInDatabase = true;
        return $this;
    }
    public function change(): static
    {
        $this->modified = true;
        return $this;
    }
    public function delete(): static
    {
        $this->deleted = true;
        return $this;
    }
    public function nullable(bool $nullable): static
    {
        $this->nullable = $nullable;
        return $this;
    }

```

```
42      abstract public function getSqlDefinition(): string;
43      /**
44       * @param array<string, mixed> $description
45       */
46      abstract public static function fromArrayDescription(
            array $description): static;
47      abstract protected function getNullableSqlDefinition()
            : string;
48  }
```

Listing 10.8: Klasa BaseColumn

Nota autora

Zwróć uwagę na flagi `modified`, `deleted` i `existsInDatabase`. Pozwalają one na rozróżnienie między nowymi kolumnami (do dodania), istniejącymi (do modyfikacji) i usuniętymi (do DROP). `SchemaBuilder` używa tych flag do generowania odpowiednich zapytań ALTER TABLE.

10.6.2 IntColumn – kolumna liczbowa

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Database\Base\Schema\Columns;
6
7   abstract class IntColumn extends BaseColumn
8   {
9       public function __construct(
10          string $name,
11          bool $nullable = true,
12          mixed $default = null,
13          public int $length = 11,
14          public bool $unsigned = false,
15          public bool $autoIncrement = false,
```

```
16          string $type = 'INT',
17      ) {
18          parent::__construct($name, $type, $nullable,
                $default);
19      }
20
21      public function unsigned(bool $unsigned): static
22      {
23          $this->unsigned = $unsigned;
24          return $this;
25      }
26
27      public function length(int $length): static
28      {
29          $this->length = $length;
30          return $this;
31      }
32
33      public function default(mixed $default): static
34      {
35          $this->default = $default;
36          return $this;
37      }
38
39      public function autoIncrement(bool $autoIncrement):
            static
40      {
41          $this->autoIncrement = $autoIncrement;
42          return $this;
43      }
44
45      abstract protected function getUnsignedSqlDefinition()
            : string;
46      abstract protected function
            getAutoIncrementSqlDefinition(): string;
47      abstract protected function
            getDefaultValueSqlDefinition(): string;
48  }
```

Listing 10.9: Klasa IntColumn

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema\ColumnDefinitions;

class IntColumn extends \Larafony\Framework\Database\Base\Schema\Columns\IntColumn
{
    public function getSqlDefinition(): string
    {
        $sql = "{$this->name} {$this->type}({$this->length})";
        $sql .= $this->getUnsignedSqlDefinition();
        $sql .= $this->getNullableSqlDefinition();
        $sql .= $this->getDefaultValueSqlDefinition();
        return $sql . $this->getAutoIncrementSqlDefinition();
    }

    /**
     * @param array<string, mixed> $description
     */
    public static function fromArrayDescription(array $description): static
    {
        preg_match('/int\((\d+)\)/i', $description['Type'], $matches);
        $length = $matches[1] ?? 11;
        $unsigned = str_contains($description['Type'], 'unsigned');
        $autoIncrement = $description['Extra'] === 'auto_increment';

        // Extract base type without length
        preg_match('/^([a-z]+)/i', $description['Type'], $typeMatches);
        $baseType = strtoupper($typeMatches[1] ?? 'INT');

        return new IntColumn(
```

```
33              $description['Field'],
34              $description['Null'] === 'YES',
35              $description['Default'],
36              (int) $length,
37              $unsigned,
38              $autoIncrement,
39              $baseType
40          );
41      }
42
43      protected function getNullableSqlDefinition(): string
44      {
45          return $this->nullable ? ' NULL' : ' NOT NULL';
46      }
47
48      protected function getUnsignedSqlDefinition(): string
49      {
50          return $this->unsigned ? ' UNSIGNED' : '';
51      }
52
53      protected function getAutoIncrementSqlDefinition():
            string
54      {
55          return $this->autoIncrement ? ' AUTO_INCREMENT' :
                '';
56      }
57
58      protected function getDefaultValueSqlDefinition():
            string
59      {
60          return $this->default !== null ? " DEFAULT {$this
                ->default}" : '';
61      }
62  }
```

Listing 10.10: Klasa IntColumn

Nota autora

Zwróć uwagę że bazowa kolumna int definiuje to co nie zależy od składni SQL (np że usingned =true ma ustawić usigned i tyle) a kolumna docelowa z mysql buduje definicję kolumny w tym języku

10.6.3 StringColumn – kolumna tekstowa

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema
    \ColumnDefinitions;

class StringColumn extends \Larafony\Framework\Database\
    Base\Schema\Columns\StringColumn
{
    public function getSqlDefinition(): string
    {
        $sql = "{$this->name} {$this->type}({$this->length
            }) ";
        $sql .= $this->getNullableSqlDefinition();
        return trim($sql . $this->
            getDefaultValueSqlDefinition());
    }

    /**
     * @param array<string, mixed> $description
     */
    public static function fromArrayDescription(array
        $description): static
    {
        preg_match('/varchar|char\((\d+)\)/i',
            $description['Type'], $matches);
        $length = $matches[1] ?? 255;

        // Extract base type without length
```

```
25          preg_match('/^([a-z]+)/i', $description['Type'],
                $typeMatches);
26          $baseType = strtoupper($typeMatches[1] ?? 'VARCHAR
                ');
27
28          return new StringColumn(
29              $description['Field'],
30              $description['Null'] === 'YES',
31              $description['Default'],
32              (int) $length,
33              $baseType
34          );
35      }
36
37      protected function getNullableSqlDefinition(): string
38      {
39          return $this->nullable ? 'NULL ' : 'NOT NULL ';
40      }
41
42      protected function getDefaultValueSqlDefinition():
            string
43      {
44          return $this->default !== null ? "DEFAULT '{$this
                ->default}' " : '';
45      }
46  }
```

Listing 10.11: Klasa StringColumn

Nota autora

Klasa bazowa wygląda niemal identycznie jak w przypadku inta - spróbuj sam lub pobierz z repo (zalecam pierwsze)

10.6.4 DateTimeColumn – kolumna daty i czasu

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema\ColumnDefinitions;

class DateTimeColumn extends \Larafony\Framework\Database\Base\Schema\Columns\DateTimeColumn
{
    public function getSqlDefinition(): string
    {
        $sql = "{$this->name} {$this->type} ";
        $sql .= $this->getNullableSqlDefinition();
        $sql .= $this->getDefaultValueDefinition();
        return trim($sql . $this->getOnUpdateDefinition());
    }

    public function getDefaultValueDefinition(): string
    {
        return $this->default !== null ? "DEFAULT {$this->default} " : '';
    }

    public function getOnUpdateDefinition(): string
    {
        return $this->onUpdate !== null ? 'ON UPDATE CURRENT_TIMESTAMP' : '';
    }

    /**
     * @param array<string, mixed> $description
     */
    public static function fromArrayDescription(array $description): static
    {
        $onUpdate = str_contains($description['Extra'] ?? '', 'on update') ? $description['Extra'] : null;
```

```
33          $precision = 0; // MySQL DESCRIBE doesn't return
               precision, would need to parse from Type
34
35          return new DateTimeColumn(
36              $description['Field'],
37              $description['Null'] === 'YES',
38              $description['Default'],
39              $onUpdate,
40              $precision,
41              $description['Type'],
42          );
43      }
44
45      protected function getNullableSqlDefinition(): string
46      {
47          return $this->nullable ? 'NULL ' : 'NOT NULL ';
48      }
49  }
```

Listing 10.12: Klasa DateTimeColumn

10.6.5 EnumColumn – kolumna z predefiniowanymi wartościami

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Database\Drivers\MySQL\Schema
       \ColumnDefinitions;
6
7   class EnumColumn extends \Larafony\Framework\Database\Base
       \Schema\Columns\EnumColumn
8   {
9       public function getSqlDefinition(): string
10      {
```

```
        $values = implode(',', array_map(static fn ($v) =>
            "'{$v}'", $this->values));
        $sql = "{$this->name} {$this->type}({$values}) ";
        $sql .= $this->getNullableSqlDefinition();
        return trim($sql . $this->getDefaultDefinition());
    }

    /**
     * @param array<string, mixed> $description
     */
    public static function fromArrayDescription(array
       $description): static
    {
        preg_match('/enum\((.*)\)/i', $description['Type'
           ], $matches);
        $values = str_getcsv($matches[1] ?? '', escape: ''
           );
        $values = array_filter($values, static fn (string|
           null $value) => (bool) $value);
        $values = array_map(
            static fn (string $value) => str_replace("'",
               '', $value),
            $values
        );
        return new EnumColumn(
            $description['Field'],
            $values,
            $description['Null'] === 'YES',
            $description['Default']
        );
    }

    protected function getNullableSqlDefinition(): string
    {
        return $this->nullable ? 'NULL ' : 'NOT NULL ';
    }

    protected function getDefaultDefinition(): string
    {
        return $this->default !== null ? "DEFAULT '{$this
           ->default}' " : '';
```

```
45        }
46    }
```

Listing 10.13: Klasa EnumColumn

Nota autora

W przypadku typu ENUM walidacja domyślnej wartości jest kluczowa. Jeśli ustawisz wartość domyślną spoza listy dozwolonych wartości, błąd pojawi się dopiero przy wykonaniu migracji – a wtedy debug jest znacznie trudniejszy. Walidacja w konstruktorze wykrywa błąd natychmiast.

10.7 Indeksy

MySQL obsługuje trzy podstawowe typy indeksów:

- **PRIMARY KEY** – unikalny identyfikator wiersza, jeden na tabelę
- **UNIQUE** – zapewnia unikalność wartości w kolumnie
- **INDEX** – przyspiesza wyszukiwanie, bez gwarancji unikalności

10.7.1 Bazowa klasa IndexDefinition

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Larafony\Framework\Database\Base\Schema\
       IndexDefinitions;
6
7  abstract readonly class IndexDefinition
8  {
```

```php
    public string $columns;

    public string $indexName;

    /**
     * @param array<int, string>|string $columns
     */
    public function __construct(
        public string $tableName,
        array|string $columns,
        ?string $indexName = null,
        string $type = 'index'
    ) {
        $this->indexName = $indexName ?? $this->generateIndexName($type, $columns);
        $this->columns = is_array($columns) ? implode(', ', $columns) : $columns;
    }

    abstract public function getSqlDefinition(): string;

    /**
     * Get inline SQL definition for use within CREATE TABLE statement
     */
    abstract public function getInlineSqlDefinition(): string;

    /**
     * @param array<int, string>|string $columns
     */
    protected function generateIndexName(string $type, array|string $columns): string
    {
        $columns = is_array($columns) ? implode('_', $columns) : $columns;
        $index = strtolower($this->tableName . '_' . $columns . '_' . $type);

        return str_replace(['-', '.'], '_', $index);
    }
```

```
43 }
```

Listing 10.14: Klasa IndexDefinition

10.7.2 PrimaryIndex

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Base\Schema\
    IndexDefinitions;

abstract readonly class PrimaryIndex extends
    IndexDefinition
{
}
```

Listing 10.15: Klasa PrimaryIndex

Nota autora

Indeksy mają dwie formy SQL: `getSqlDefinition()` dla ALTER TABLE i `getInlineSqlDefinition()` dla CREATE TABLE. W CREATE TABLE indeksy są częścią definicji tabeli, w ALTER TABLE są osobnymi poleceniami.

10.8 SchemaBuilder – orkiestrator

Klasa `SchemaBuilder` łączy wszystkie elementy w jedną całość. Przyjmuje callback definiujący strukturę tabeli, kompiluje go przez `Grammar` i opcjonalnie wykonuje wygenerowany SQL.

10.8.1 Bazowa klasa SchemaBuilder

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Base\Schema;

use Closure;
use Larafony\Framework\Database\Base\Contracts\
    ConnectionContract;
use Larafony\Framework\Database\Base\Contracts\
    SchemaBuilderContract;
use Larafony\Framework\Database\Drivers\MySQL\Grammar;

abstract class SchemaBuilder implements
    SchemaBuilderContract
{
    protected Grammar $grammar;
    public function __construct(protected readonly
        ConnectionContract $connection)
    {
    }

    #[\NoDiscard]
    abstract public function create(string $table, Closure
        $callback): string;
    #[\NoDiscard]
    abstract public function table(string $table, Closure
        $callback): string;
    #[\NoDiscard]
    abstract public function drop(string $table): string;
    #[\NoDiscard]
    abstract public function dropIfExists(string $table):
        string;

    public function execute(string $sql): bool
    {
        $this->connection->query($sql);
        return true;
```

```
32      }
33
34      /**
35       * @return array<int, string>
36       */
37      abstract public function getColumnListing(string
           $table): array;
38  }
```

Listing 10.16: Klasa SchemaBuilder (bazowa)

10.8.2 MySQL SchemaBuilder

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Database\Drivers\MySQL;
6
7   use Closure;
8   use Larafony\Framework\Database\Base\Contracts\
       ConnectionContract;
9   use Larafony\Framework\Database\Drivers\MySQL\Schema\
       DatabaseInfo;
10  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       TableDefinition;
11
12  class SchemaBuilder extends \Larafony\Framework\Database\
       Base\Schema\SchemaBuilder
13  {
14      public function __construct(ConnectionContract
           $connection)
15      {
16          parent::__construct($connection);
17          $this->grammar = new Grammar();
18      }
19
```

```
20      #[\NoDiscard]
21      public function create(string $table, Closure
           $callback): string
22      {
23          $tableDefinition = new TableDefinition($table);
24          $callback($tableDefinition);
25          return $this->grammar->compileCreate(
               $tableDefinition);
26      }
27
28      #[\NoDiscard]
29      public function table(string $table, Closure $callback
           ): string
30      {
31          $tableDefinition = new DatabaseInfo($this->
               connection)->getTable($table);
32          $callback($tableDefinition);
33
34          $statements = array_filter([
35              $this->grammar->compileAddColumns(
                   $tableDefinition),
36              $this->grammar->compileModifyColumns(
                   $tableDefinition),
37              $this->grammar->compileDropColumns(
                   $tableDefinition),
38          ]);
39
40          return implode(';' . PHP_EOL, $statements);
41      }
42
43      #[\NoDiscard]
44      public function drop(string $table): string
45      {
46          return $this->grammar->compileDropTable($table);
47      }
48
49      #[\NoDiscard]
50      public function dropIfExists(string $table): string
51      {
52          return $this->grammar->compileDropTable($table,
               ifExists: true);
```

```
53      }
54
55      /**
56       * @return array<int, string>
57       */
58      #[\NoDiscard]
59      public function getColumnListing(string $table): array
60      {
61          return new DatabaseInfo($this->connection)->
                getTable($table)->columnNames;
62      }
63  }
```

Listing 10.17: Klasa MySQL SchemaBuilder

Nota autora

Zwróć uwagę na atrybut `#[NoDiscard]` przy metodach `create()`, `table()`, itp. To świadoma decyzja projektowa – metody te **jedynie zwracają string SQL**, nie wykonują żadnych zmian na bazie danych. Dlaczego tak? Bo rozdzielenie generowania SQL od jego wykonania daje elastyczność:

```
1  // Podgląd wygenerowanego SQL (np. do debugowania)
2  $sql = Schema::create('users', fn($t) => $t->id());
3  dump($sql);
4
5  // Wykonanie SQL
6  Schema::execute($sql);
7
8  // Lub elegancko z pipe operatorem (PHP 8.5)
9  Schema::create('users', fn($t) => $t->id()) |> Schema::
       execute(...);
```

`#[NoDiscard]` informuje analizatory statyczne (PHPStan, Psalm) i IDE, że ignorowanie zwracanej wartości to prawdopodobnie błąd – programista

zapomniał coś zrobić z wygenerowanym SQL. To kolejny przykład "fail fast" – błąd wykrywamy przy analizie statycznej, nie przy debugowaniu pustej bazy danych.

10.9 Buildery SQL

Kompilacja `TableDefinition` do SQL jest delegowana do wyspecjalizowanych builderów. To kluczowy element wzorca Grammar – buildery są "tłumaczami", które przekształcają abstrakcyjną definicję tabeli w konkretny SQL dla danego silnika bazy danych.

Nota autora

Dlaczego wzorzec Grammar jest elegancki? Bo `TableDefinition` nie wie nic o SQL. To czysta struktura danych opisująca *co* chcemy osiągnąć. Buildery wiedzą *jak* to zrobić dla konkretnej bazy. Dzięki temu:

- Ten sam kod migracji działa na MySQL, PostgreSQL, SQLite
- Łatwo dodać wsparcie dla nowej bazy – piszesz nowe buildery, nie zmieniasz istniejącego kodu
- Testy są trywialne – przekazujesz `TableDefinition`, sprawdzasz wygenerowany string

To realizacja zasady Open/Closed z SOLID: system jest otwarty na rozszerzenia (nowe bazy), zamknięty na modyfikacje (istniejący kod nie wymaga zmian).

10.9.1 CreateTableBuilder

`CreateTableBuilder` generuje pełną instrukcję `CREATE TABLE`. Sprytnie wykorzystuje fakt, że każda kolumna w `TableDefinition` potrafi sama wygenerować swoją definicję SQL poprzez metodę `getSqlDefinition()`:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema\Builders;

use Larafony\Framework\Database\Base\Schema\Columns\BaseColumn;
use Larafony\Framework\Database\Base\Schema\IndexDefinitions\IndexDefinition;
use Larafony\Framework\Database\Base\Schema\TableDefinition;

class CreateTableBuilder extends \Larafony\Framework\Database\Base\Schema\Builders\Builders\CreateTableBuilder
{
    public function build(TableDefinition $table): string
    {
        $tableName = $table->tableName;
        $columns = $table->columns;
        $columnDefinitions = array_map(static fn (BaseColumn $column) => $column->getSqlDefinition(), $columns);

        // Add inline index definitions to CREATE TABLE
        $inlineIndexes = $this->getInlineIndexes($table);

        $allDefinitions = array_merge($columnDefinitions, $inlineIndexes);

        return sprintf(
            'CREATE TABLE %s (%s);',
            $tableName,
            implode(', ', $allDefinitions),
        );
    }

    /**
     * Get inline index definitions for CREATE TABLE
```

```
         statement
33       *
34       * @return array<int, string>
35       */
36      private function getInlineIndexes(TableDefinition
            $table): array
37      {
38          $indexes = $table->indexes;
39          return array_map(
40              static fn (IndexDefinition $index) => $index->
                    getInlineSqlDefinition(),
41              $indexes,
42          );
43      }
44  }
```

Listing 10.18: Klasa CreateTableBuilder

Nota autora

Zauważ elegancję tego podejścia: builder nie musi wiedzieć, czy kolumna to `INT`, `VARCHAR` czy `ENUM`. Wywołuje `getSqlDefinition()` i dostaje gotowy fragment SQL. Polimorfizm w akcji – każda klasa kolumny sama wie, jak się "zserializować" do SQL.

10.9.2 AddColumns Builder

`AddColumns` generuje instrukcje `ALTER TABLE ... ADD COLUMN`. Wykorzystuje flagę `existsInDatabase` w kolumnach – przetwarza tylko te, które nie istnieją jeszcze w bazie:

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Database\Drivers\MySQL\Schema
```

```
    \Builders;
6
7   use Larafony\Framework\Database\Base\Schema\Columns\
        BaseColumn;
8   use Larafony\Framework\Database\Base\Schema\
        TableDefinition;
9
10  class AddColumns extends \Larafony\Framework\Database\Base
        \Schema\Builders\Builders\AddColumns
11  {
12      #[\NoDiscard]
13      public function build(TableDefinition $table): string
14      {
15          $columns = $table->columns;
16          $columns = array_filter(
17              $columns,
18              static fn (BaseColumn $column) => ! $column->
                    existsInDatabase && ! $column->modified &&
                    ! $column->deleted
19          );
20          if ($columns === []) {
21              return '';
22          }
23          $tableName = $table->tableName;
24          $definitions = array_map(
25              static fn (BaseColumn $column) => 'ADD COLUMN
                    ' .$column->getSqlDefinition(),
26              $columns
27          );
28          return sprintf('ALTER TABLE %s %s;', $tableName,
                implode(', ', $definitions));
29      }
30  }
```

Listing 10.19: Klasa AddColumns

10.9.3 ChangeColumns Builder

`ChangeColumns` generuje instrukcje `ALTER TABLE ... MODIFY COLUMN`. Przetwarza tylko kolumny oznaczone flagą `modified`:

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Database\Drivers\MySQL\Schema\Builders;

use Larafony\Framework\Database\Base\Schema\Columns\BaseColumn;
use Larafony\Framework\Database\Base\Schema\TableDefinition;

class ChangeColumns extends \Larafony\Framework\Database\Base\Schema\Builders\Builders\ChangeColumns
{
    #[\NoDiscard]
    public function build(TableDefinition $table): string
    {
        $columns = $table->columns;
        $columns = array_filter(
            $columns,
            static fn (BaseColumn $column) => $column->modified
        );
        if ($columns === []) {
            return '';
        }
        $tableName = $table->tableName;
        $definitions = array_map(
            static fn (BaseColumn $column) => 'MODIFY COLUMN ' . $column->getSqlDefinition(),
            $columns
        );
        return sprintf('ALTER TABLE %s %s;', $tableName, implode(', ', $definitions));
```

```
29      }
30  }
```

Listing 10.20: Klasa ChangeColumns

10.9.4 DropColumns Builder

`DropColumns` generuje instrukcje `ALTER TABLE ... DROP COLUMN`. Przetwarza tylko kolumny oznaczone flagą `deleted`:

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Database\Drivers\MySQL\Schema
        \Builders;
6
7   use Larafony\Framework\Database\Base\Schema\Columns\
        BaseColumn;
8   use Larafony\Framework\Database\Base\Schema\
        TableDefinition;
9
10  class DropColumns extends \Larafony\Framework\Database\
        Base\Schema\Builders\Builders\DropColumns
11  {
12      #[\NoDiscard]
13      public function build(TableDefinition $table): string
14      {
15          $columns = $table->columns;
16          $columns = array_filter(
17              $columns,
18              static fn (BaseColumn $column) => $column->
                    deleted
19          );
20          if ($columns === []) {
21              return '';
22          }
```

```
23          $tableName = $table->tableName;
24          $modifiedColumns = array_map(
25              static fn (BaseColumn $column) => 'DROP COLUMN
                    ' .$column->name,
26              $columns
27          );
28          return 'ALTER TABLE ' .$tableName. ' ' .implode(',
                ', $modifiedColumns);
29      }
30  }
```

Listing 10.21: Klasa DropColumns

Nota autora

Zwróć uwagę na spójność architektury: wszystkie buildery działają na tym samym `TableDefinition`, ale każdy patrzy na inne flagi. To pozwala na eleganckie rozdzielenie operacji ALTER TABLE na trzy niezależne części. Gdy wywołujesz `$schema->table('users', fn($t) => ...)`, `SchemaBuilder`:

1. Pobiera aktualną strukturę tabeli z bazy (metoda `getTable()`)
2. Wykonuje callback, który modyfikuje `TableDefinition`
3. Wywołuje wszystkie trzy buildery i łączy niepuste wyniki

Dzięki temu jeden callback może jednocześnie dodawać, modyfikować i usuwać kolumny – a system sam wygeneruje odpowiednie instrukcje SQL.

10.10 Testy jednostkowe

10.10.1 Testy klasy Grammar

```
1  <?php
```

```php

declare(strict_types=1);

namespace Larafony\Framework\Tests\Database\MySQL;

use Larafony\Framework\Database\Drivers\MySQL\Grammar;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\IntColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    ColumnDefinitions\StringColumn;
use Larafony\Framework\Database\Drivers\MySQL\Schema\
    TableDefinition;
use Larafony\Framework\Tests\TestCase;

class GrammarTest extends TestCase
{
    private Grammar $grammar;

    protected function setUp(): void
    {
        parent::setUp();
        $this->grammar = new Grammar();
    }

    public function testCompilesCreateTable(): void
    {
        $table = new TableDefinition('users', [
            'id' => new IntColumn('id', type: 'INT'),
            'name' => new StringColumn('name', length:
                255, type: 'VARCHAR'),
        ]);

        $sql = $this->grammar->compileCreate($table);

        $this->assertStringContainsString('CREATE TABLE
            users', $sql);
        $this->assertStringContainsString('id', $sql);
        $this->assertStringContainsString('name', $sql);
    }

    public function testCompilesAddColumns(): void
```

```
38      {
39          $table = new TableDefinition('users', [
40              'email' => new StringColumn('email', length:
                    255, type: 'VARCHAR'),
41          ]);
42
43          $sql = $this->grammar->compileAddColumns($table);
44
45          $this->assertStringContainsString('ALTER TABLE
                users', $sql);
46          $this->assertStringContainsString('ADD COLUMN',
                $sql);
47          $this->assertStringContainsString('email', $sql);
48      }
49
50      public function
            testCompilesAddColumnsReturnsEmptyForNoColumns():
            void
51      {
52          $table = new TableDefinition('users', []);
53
54          $sql = $this->grammar->compileAddColumns($table);
55
56          $this->assertEmpty($sql);
57      }
58
59      public function testCompilesModifyColumns(): void
60      {
61          $emailColumn = new StringColumn('email', length:
                100, type: 'VARCHAR');
62          $emailColumn->change();
63
64          $table = new TableDefinition('users', [
65              'email' => $emailColumn,
66          ]);
67
68          $sql = $this->grammar->compileModifyColumns($table
                );
69
70          $this->assertStringContainsString('ALTER TABLE
                users', $sql);
```

```
71          $this->assertStringContainsString('MODIFY COLUMN',
                $sql);
72          $this->assertStringContainsString('email', $sql);
73      }
74
75      public function
            testCompilesModifyColumnsReturnsEmptyForNoModifiedColumns
            (): void
76      {
77          $table = new TableDefinition('users', [
78              'email' => new StringColumn('email', length:
                    255, type: 'VARCHAR'),
79          ]);
80
81          $sql = $this->grammar->compileModifyColumns($table
                );
82
83          $this->assertEmpty($sql);
84      }
85
86      public function testCompilesDropColumns(): void
87      {
88          $emailColumn = new StringColumn('email', length:
                255, type: 'VARCHAR');
89          $emailColumn->delete();
90
91          $table = new TableDefinition('users', [
92              'email' => $emailColumn,
93          ]);
94
95          $sql = $this->grammar->compileDropColumns($table);
96
97          $this->assertStringContainsString('ALTER TABLE
                users', $sql);
98          $this->assertStringContainsString('DROP COLUMN
                email', $sql);
99      }
100
101     public function
            testCompilesDropColumnsReturnsEmptyForNoDeletedColumns
            (): void
```

```
102     {
103         $table = new TableDefinition('users', [
104             'email' => new StringColumn('email', length:
                    255, type: 'VARCHAR'),
105         ]);
106 
107         $sql = $this->grammar->compileDropColumns($table);
108 
109         $this->assertEmpty($sql);
110     }
111 
112     public function testCompilesDropTable(): void
113     {
114         $sql = $this->grammar->compileDropTable('users',
                false);
115 
116         $this->assertStringContainsString('DROP TABLE',
                $sql);
117         $this->assertStringContainsString('users', $sql);
118     }
119 
120     public function testCompilesDropTableIfExists(): void
121     {
122         $sql = $this->grammar->compileDropTable('users',
                true);
123 
124         $this->assertStringContainsString('DROP TABLE',
                $sql);
125         $this->assertStringContainsString('IF EXISTS',
                $sql);
126         $this->assertStringContainsString('users', $sql);
127     }
128 }
```

Listing 10.22: GrammarTest

10.10.2 Testy IntColumn

```php
<?php

declare(strict_types=1);

namespace Larafony\Framework\Tests\Database\MySQL;

use Larafony\Framework\Database\Drivers\MySQL\Schema\
   ColumnDefinitions\IntColumn;
use Larafony\Framework\Tests\TestCase;

class IntColumnTest extends TestCase
{
    public function testCreatesIntColumn(): void
    {
        $column = new IntColumn('id', type: 'INT');

        $this->assertEquals('id', $column->name);
        $this->assertEquals('INT', $column->type);
    }

    public function testGeneratesSqlDefinition(): void
    {
        $column = new IntColumn('age', type: 'INT');
        $sql = $column->getSqlDefinition();

        $this->assertStringContainsString('age', $sql);
        $this->assertStringContainsString('INT', $sql);
    }

    public function testSupportsLength(): void
    {
        $column = new IntColumn('count', type: 'INT');
        $column = $column->length(10);
        $sql = $column->getSqlDefinition();

        $this->assertStringContainsString('INT(10)', $sql)
           ;
    }

    public function testSupportsUnsigned(): void
    {
```

```
40          $column = new IntColumn('count', type: 'INT');
41          $column = $column->unsigned(true);
42          $sql = $column->getSqlDefinition();
43
44          $this->assertStringContainsString('UNSIGNED', $sql
                );
45      }
46
47      public function testSupportsAutoIncrement(): void
48      {
49          $column = new IntColumn('id', type: 'INT');
50          $column = $column->autoIncrement(true);
51          $sql = $column->getSqlDefinition();
52
53          $this->assertStringContainsString('AUTO_INCREMENT'
                , $sql);
54      }
55
56      public function testSupportsNullable(): void
57      {
58          $column = new IntColumn('optional', type: 'INT');
59          $column = $column->nullable(true);
60          $sql = $column->getSqlDefinition();
61
62          $this->assertStringContainsString('NULL', $sql);
63      }
64
65      public function testSupportsNotNullable(): void
66      {
67          $column = new IntColumn('required', type: 'INT');
68          $column = $column->nullable(false);
69          $sql = $column->getSqlDefinition();
70
71          $this->assertStringContainsString('NOT NULL', $sql
                );
72      }
73
74      public function testSupportsDefault(): void
75      {
76          $column = new IntColumn('status', type: 'INT');
77          $column = $column->default(0);
```

```
78          $sql = $column->getSqlDefinition();
79
80          $this->assertStringContainsString('DEFAULT', $sql)
                ;
81          $this->assertStringContainsString('0', $sql);
82      }
83  }
```

Listing 10.23: IntColumnTest

10.10.3 Testy TableDefinition

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace Larafony\Framework\Tests\Database\MySQL;
6
7   use Larafony\Framework\Database\Drivers\MySQL\Schema\
       ColumnDefinitions\DateTimeColumn;
8   use Larafony\Framework\Database\Drivers\MySQL\Schema\
       ColumnDefinitions\IntColumn;
9   use Larafony\Framework\Database\Drivers\MySQL\Schema\
       ColumnDefinitions\StringColumn;
10  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       ColumnDefinitions\TextColumn;
11  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       IndexDefinitions\NormalIndex;
12  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       IndexDefinitions\PrimaryIndex;
13  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       IndexDefinitions\UniqueIndex;
14  use Larafony\Framework\Database\Drivers\MySQL\Schema\
       TableDefinition;
15  use Larafony\Framework\Tests\TestCase;
16
17  class TableDefinitionTest extends TestCase
```

```
{
    public function testCreatesTableDefinition(): void
    {
        $table = new TableDefinition('users');

        $this->assertEquals('users', $table->tableName);
        $this->assertEmpty($table->columns);
    }

    public function testIntegerReturnsIntColumn(): void
    {
        $table = new TableDefinition('users');

        $column = $table->integer('age');

        $this->assertInstanceOf(IntColumn::class, $column)
            ;
        $this->assertEquals('age', $column->name);
        $this->assertEquals('INT', $column->type);
    }

    public function
        testBigIntegerReturnsIntColumnWithLength(): void
    {
        $table = new TableDefinition('users');

        $column = $table->bigInteger('id');

        $this->assertInstanceOf(IntColumn::class, $column)
            ;
        $this->assertEquals('id', $column->name);
    }

    public function
        testSmallIntegerReturnsIntColumnWithLength(): void
    {
        $table = new TableDefinition('users');

        $column = $table->smallInteger('count');

```

```
54          $this->assertInstanceOf(IntColumn::class, $column)
                ;
55          $this->assertEquals('count', $column->name);
56      }
57
58      public function testStringReturnsStringColumn(): void
59      {
60          $table = new TableDefinition('users');
61
62          $column = $table->string('name', 100);
63
64          $this->assertInstanceOf(StringColumn::class,
                $column);
65          $this->assertEquals('name', $column->name);
66          $this->assertEquals('VARCHAR', $column->type);
67      }
68
69      public function
            testCharReturnsStringColumnWithCharType(): void
70      {
71          $table = new TableDefinition('users');
72
73          $column = $table->char('code', 5);
74
75          $this->assertInstanceOf(StringColumn::class,
                $column);
76          $this->assertEquals('code', $column->name);
77          $this->assertEquals('CHAR', $column->type);
78      }
79
80      public function testTextReturnsTextColumn(): void
81      {
82          $table = new TableDefinition('posts');
83
84          $column = $table->text('content');
85
86          $this->assertInstanceOf(TextColumn::class, $column
                );
87          $this->assertEquals('content', $column->name);
88          $this->assertEquals('TEXT', $column->type);
89      }
```

```
90
91      public function testMediumTextReturnsTextColumn():
            void
92      {
93          $table = new TableDefinition('posts');
94
95          $column = $table->mediumText('description');
96
97          $this->assertInstanceOf(TextColumn::class, $column
                );
98          $this->assertEquals('description', $column->name);
99          $this->assertEquals('MEDIUMTEXT', $column->type);
100     }
101
102     public function testLongTextReturnsTextColumn(): void
103     {
104         $table = new TableDefinition('posts');
105
106         $column = $table->longText('document');
107
108         $this->assertInstanceOf(TextColumn::class, $column
                );
109         $this->assertEquals('document', $column->name);
110         $this->assertEquals('LONGTEXT', $column->type);
111     }
112
113     public function testJsonReturnsTextColumn(): void
114     {
115         $table = new TableDefinition('settings');
116
117         $column = $table->json('data');
118
119         $this->assertInstanceOf(TextColumn::class, $column
                );
120         $this->assertEquals('data', $column->name);
121         $this->assertEquals('JSON', $column->type);
122     }
123
124     public function testDateTimeReturnsDateTimeColumn():
            void
125     {
```

```
126         $table = new TableDefinition('logs');
127
128         $column = $table->dateTime('created_at');
129
130         $this->assertInstanceOf(DateTimeColumn::class,
                $column);
131         $this->assertEquals('created_at', $column->name);
132         $this->assertEquals('DATETIME', $column->type);
133     }
134
135     public function testTimestampReturnsDateTimeColumn():
            void
136     {
137         $table = new TableDefinition('logs');
138
139         $column = $table->timestamp('updated_at');
140
141         $this->assertInstanceOf(DateTimeColumn::class,
                $column);
142         $this->assertEquals('updated_at', $column->name);
143         $this->assertEquals('TIMESTAMP', $column->type);
144     }
145
146     public function testTimeReturnsDateTimeColumn(): void
147     {
148         $table = new TableDefinition('schedules');
149
150         $column = $table->time('start_time');
151
152         $this->assertInstanceOf(DateTimeColumn::class,
                $column);
153         $this->assertEquals('start_time', $column->name);
154         $this->assertEquals('TIME', $column->type);
155     }
156
157     public function testDateReturnsDateTimeColumn(): void
158     {
159         $table = new TableDefinition('events');
160
161         $column = $table->date('event_date');
162
```

```
163         $this->assertInstanceOf(DateTimeColumn::class,
                $column);
164         $this->assertEquals('event_date', $column->name);
165         $this->assertEquals('DATE', $column->type);
166     }
167
168     public function testIndexReturnsNormalIndex(): void
169     {
170         $table = new TableDefinition('users');
171
172         $index = $table->index('email');
173
174         $this->assertInstanceOf(NormalIndex::class, $index
                );
175         $this->assertEquals('users', $index->tableName);
176         $this->assertEquals('email', $index->columns);
177     }
178
179     public function testIndexWithCustomName(): void
180     {
181         $table = new TableDefinition('users');
182
183         $index = $table->index('email', 'idx_users_email')
                ;
184
185         $this->assertInstanceOf(NormalIndex::class, $index
                );
186         $this->assertEquals('idx_users_email', $index->
                indexName);
187     }
188
189     public function testPrimaryReturnsPrimaryIndex(): void
190     {
191         $table = new TableDefinition('users');
192
193         $index = $table->primary('id');
194
195         $this->assertInstanceOf(PrimaryIndex::class,
                $index);
196         $this->assertEquals('users', $index->tableName);
197         $this->assertEquals('id', $index->columns);
```

```
198     }
199
200     public function testPrimaryWithCompositeKey(): void
201     {
202         $table = new TableDefinition('user_roles');
203
204         $index = $table->primary(['user_id', 'role_id']);
205
206         $this->assertInstanceOf(PrimaryIndex::class,
                $index);
207     }
208
209     public function testUniqueReturnsUniqueIndex(): void
210     {
211         $table = new TableDefinition('users');
212
213         $index = $table->unique('email');
214
215         $this->assertInstanceOf(UniqueIndex::class, $index
            );
216         $this->assertEquals('users', $index->tableName);
217         $this->assertEquals('email', $index->columns);
218     }
219
220     public function testUniqueWithCustomName(): void
221     {
222         $table = new TableDefinition('users');
223
224         $index = $table->unique('email', '
            unique_user_email');
225
226         $this->assertInstanceOf(UniqueIndex::class, $index
            );
227         $this->assertEquals('unique_user_email', $index->
            indexName);
228     }
229
230     public function testSupportsMethodChaining(): void
231     {
232         $table = new TableDefinition('users');
233
```

```
234         $column = $table->string('email', 255);
235
236         // Verify method chaining is supported on returned
                columns
237         $this->assertInstanceOf(StringColumn::class,
                $column);
238     }
239 }
```

Listing 10.24: TableDefinitionTest

Nota autora

Testy kolumn sprawdzają wszystkie kombinacje opcji: nullable/not null, default values, auto_increment, unsigned. To ważne, bo błąd w generowaniu SQL może być trudny do wykrycia – baza danych nie zawsze rzuci błędem, czasem po prostu zinterpretuje zapytanie inaczej niż zamierzaliśmy.

10.11 Podsumowanie

W tym rozdziale zbudowaliśmy fundament DBAL – warstwy abstrakcji nad bazą danych:

- Interfejs `ConnectionContract` definiujący połączenie z bazą
- Wzorzec `Grammar` do kompilacji obiektowych definicji do SQL
- `TableDefinition` z fluent API do definiowania tabel
- Klasy kolumn: `IntColumn`, `StringColumn`, `DateTimeColumn`, `EnumColumn`
- Klasy indeksów: `PrimaryIndex`, `UniqueIndex`, `NormalIndex`
- `SchemaBuilder` jako główny orkiestrator operacji na schemacie
- Buildery SQL: `CreateTableBuilder`, `AddColumns`, `ChangeColumns`, `DropColumns`

Nota autora

Architektura oparta na `Grammar` może wydawać się bardziej skomplikowana niż proste generowanie SQL w jednej klasie. Ale ta inwestycja zwraca się wielokrotnie:

- Dodanie wsparcia dla PostgreSQL to nowy katalog w `Drivers/`, nie modyfikacja istniejącego kodu
- Testy są prostsze – możemy testować `Grammar` niezależnie od połączenia z bazą
- Fluent API jest czytelne dla programistów znających Laravela

W kolejnym rozdziale zajmiemy się `QueryBuilder` – klasą do budowania zapytań SELECT, INSERT, UPDATE i DELETE.
Czas na zapytania!

Spis listingów

Skorowidz

Skorowidz