DAMIAN JÓŹWIAK

Współczesny SOLIDny FRAMEWORK, PHP

PHP 8.4 W PRAKTYCE: TWORZENIE APLIKACJI I FRAMEWORKÓW NOWEJ GENERACJI

PHP 8.4 w praktyce: Tworzenie aplikacji i frameworków nowej generacji

Autor: **Damian Jóźwiak**

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

Pierwsze wydanie: **2024**

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

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

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. Framework został opracowany głównie jako materiał dydaktyczny, a jego zastosowanie w warunkach produkcyjnych wiąże się z ryzykiem, które każdy użytkownik podejmuje na własną odpowiedzialność.

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. 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);

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

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

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

interface JsonContract
{
    public function json(): string;
```

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

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

Listing 1.1: Zasady SOLID

Zasada jednej odpowiedzialności - SRP

Klasa powinna mieć tylko jedną odpowiedzialność. Jedyne co potrafią klasy `Square` i `Square` 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  class Car {
6      public string $engine;
7      public int $wheels;
8      public string $color;
9  }
10
11 class CarBuilder {
12     private Car $car;
```

```
13
14      public function __construct() {
15          $this->car = new Car();
16      }
17
18      public function setEngine($engine): self
19      {
20          $this->car->engine = $engine;
21          return $this;
22      }
23
24      public function setWheels($wheels):self
25      {
26          $this->car->wheels = $wheels;
27          return $this;
28      }
29
30      public function setColor($color):self
31      {
32          $this->car->color = $color;
33          return $this;
34      }
35
36      public function build() {
37          return $this->car;
38      }
39  }
40
41  $car = new CarBuilder()
42      ->setEngine('V8')
43      ->setWheels(4)
44      ->setColor('red')
45      ->build();
```

Listing 1.2: wzorzec Budowniczy

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.

```php
<?php

declare(strict_types=1);

interface Coffee {
    public function getCost(): int;
    public function getDescription(): string;
}

class SimpleCoffee implements Coffee {
    public function getCost(): int
    {
        return 5;
    }

    public function getDescription(): string {
        return "Simple coffee";
    }
}

readonly class MilkDecorator implements Coffee {

    public function __construct(protected Coffee $coffee)
    {}

    public function getCost(): int
    {
        return $this->coffee->getCost() + 2;
    }

```

```
31      public function getDescription(): string
32      {
33          return $this->coffee->getDescription() . ", milk";
34      }
35  }
36
37  $coffee = new MilkDecorator(new SimpleCoffee());
38  echo $coffee->getCost(); // 7
39  echo $coffee->getDescription(); // Simple coffee, milk
```

Listing 1.3: wzorzec Dekorator

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.

```
1   <?php
2
3   declare(strict_types=1);
4
5   interface Animal
6   {
7       public function makeSound(): string;
8   }
9
10  class Dog implements Animal
11  {
12      public function makeSound(): string
13      {
14          return "Woof!";
15      }
16  }
17
```

```php
class Cat implements Animal
{
    public function makeSound(): string
    {
        return "Meow!";
    }
}

class AnimalFactory
{
    public static function createAnimal($type): Animal
    {
        return match ($type) {
            'dog' => new Dog(),
            'cat' => new Cat(),
        };
    }
}


echo AnimalFactory::createAnimal('dog')->makeSound(); // Woof!
echo AnimalFactory::createAnimal('cat')->makeSound(); // Meow!
```

Listing 1.4: wzorzec Budowniczy

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

```php
<?php

declare(strict_types=1);

class Engine
{
    public function start(): void
    {
        echo "Engine started\n";
    }
}

class AirConditioner
{
    public function turnOn(): void
    {
        echo "Air conditioner is on\n";
    }
}

class CarFacade
{
    private Engine $engine;
    private AirConditioner $airConditioner;

    public function __construct()
    {
        $this->engine = new Engine();
        $this->airConditioner = new AirConditioner();
    }

    public function startCar()
    {
        $this->engine->start();
        $this->airConditioner->turnOn();
    }
}

$car = new CarFacade();
$car->startCar();
```

Listing 1.5: wzorzec Fasada

1.2.5 Kompozyt

Wzorzec Kompozyt (Composite) pozwala na reprezentowanie hierarchii obiektów w postaci drzewa. Umożliwia traktowanie pojedynczych obiektów oraz grup obiektów w jednakowy sposób. Jest przydatny, gdy chcemy budować obiekty złożone, które składają się z wielu prostszych komponentów.

```php
<?php

declare(strict_types=1);

interface Component
{
    public function operation(): string;
}

class Leaf implements Component
{
    public function operation(): string
    {
        return "Leaf";
    }
}

class Composite implements Component
{
    /**
     * @var array<int, Component>
     */
    private array $children = [];

    public function add(Component $component): static
    {
        $this->children[] = $component;
        return $this;
    }

    public function operation(): string
    {
        return sprintf(
```

```
34              'Composite(%s)',
35              implode(', ', $this->children)
36          );
37      }
38  }
39
40  echo new Composite()
41      ->add(new Leaf())
42      ->add(new Leaf())
43      ->operation(); // Composite(Leaf, Leaf)
```

Listing 1.6: wzorzec Kompozyt

1.2.6 Singleton

Wzorzec Singleton zapewnia, że dana klasa ma tylko jedną instancję i udostępnia ją globalnie. Jest to szczególnie przydatne, gdy wymagamy globalnego dostępu do zasobu, na przykład połączenia z bazą danych lub logowania. Singleton zapewnia kontrolę nad tworzeniem obiektów i chroni przed wielokrotnymi instancjami. Klasy `Web/Application` oraz `Console/Application` które stworzymy w rozdziałach czwartym i szóstym będą doskonałym przykładem tego wzorca, ponieważ w całym frameworku wystarczy nam po jednej instancji tej klasy w zależności od tego czy będziemy pracować w środowisku webowym czy konsolowym

```
1   <?php
2
3   declare(strict_types=1);
4
5   class Singleton
6   {
7       private static Singleton $instance;
8
9       private function __construct()
10      {
11      }
```

```
12
13      public static function getInstance(): static
14      {
15          self::$instance ??= new Singleton();
16          return self::$instance;
17      }
18  }
19
20  $instance1 = Singleton::getInstance();
21  $instance2 = Singleton::getInstance();
22  var_dump($instance1 === $instance2); // true
```

Listing 1.7: wzorzec Singleton

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.

1.4 Zarządzanie zależnościami

Composer jest narzędziem do zarządzania zależnościami w PHP, które umożliwia deklarowanie bibliotek, od których zależy projekt, oraz zarządzanie ich instalacją i aktualizacją.

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. Jednak 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

1.5.1 Utworzenie Struktury Katalogów

Najpierw utwórzmy podstawową strukturę katalogów dla projektu. W konsoli (lub terminalu) wykonaj następujące polecenia:

```
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`:

```
[language=bash]
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 bardziej zaawansowane i czytelne wyświetlanie informacji o zmiennych w porównaniu do standardowych funkcji var_dump i print_r w PHP. var_dump i print_r wyświetlają struktury danych bez formatowania, co może być trudne do odczytania przy złożonych strukturach, takich jak wielowymiarowe tablice czy obiekty. Natomiast symfony/var-dumper formatuje dane w bardziej przejrzysty sposób, dodając kolory i strukturalne wcięcia, co ułatwia analizę i debugowanie kodu.

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

```
cd ..
touch composer.json
```

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

```
[language=JSON]
{
    "repositories": [
        {
            "type": "path",
            "url": "./framework"
        }
    ],
```

```
    "require": {
        "my_project/framework": "*",
        "symfony/var-dumper": "^5.0"
    }
}
```

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 php8.4-fpm
```

dodaj do pliku `/etc/hosts` (lub `C:\Windows\System32\drivers\etc\hosts` w systemie Windows)

```
[language=bash]
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/php8.4-fpm.sock|fcgi:/
    </FilesMatch>
    ErrorLog ${APACHE_LOG_DIR}/framework.local_error.log
    CustomLog ${APACHE_LOG_DIR}/framework.local_access.log com
</VirtualHost>
```

Aktywuj vHosta i przeładuj Apache:

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

Nginx Stwórz plik konfiguracyjny dla vHosta:

```
1 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/php8.4-fpm.sock;
    }
    error_log /var/log/nginx/framework.local_error.log;
    access_log /var/log/nginx/framework.local_access.log;
}
```

Aktywuj vHosta i przeładuj Nginx:

```
sudo ln -s /etc/nginx/sites-available/framework.local /etc/ngi
enabled/
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:^11
```

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 Xdebug i pokrycie kodu

Xdebug to rozszerzenie PHP, które umożliwia debugowanie i profilowanie kodu. W kontekście testowania jednostkowego, Xdebug jest często używany do generowania raportów pokrycia kodu.

Instalacja Xdebug

Aby zainstalować Xdebug, należy wykonać następujące kroki:

```
pecl install xdebug
```

Następnie należy dodać Xdebug do pliku ‘php.ini‘:

```
zend_extension="xdebug.so"
```

Generowanie raportu pokrycia kodu

Aby wygenerować raport pokrycia kodu, należy uruchomić PHPUnit z odpowiednimi opcjami:

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

Raport pokrycia kodu zostanie zapisany w katalogu ‘coverage‘.

Rozdział 10

DBAL - ORM

W świecie programowania aplikacji webowych, kluczową rolę odgrywają modele, które reprezentują dane i ich strukturę. Modele są abstrakcją, pozwalającą nam pracować z danymi w bardziej uporządkowany i zrozumiały sposób. Dzięki nim możemy lepiej odwzorować rzeczywiste obiekty biznesowe oraz skomplikowane relacje pomiędzy nimi, bez konieczności bezpośredniego manipulowania bazą danych. Innymi słowy, modele są "pomostem"pomiędzy światem obiektowym, w którym operujemy, a relacyjnymi bazami danych, z których korzystają nasze aplikacje.

Modele są nieodzowne, ponieważ upraszczają proces budowania aplikacji, pozwalając na łatwiejsze zarządzanie danymi oraz ich relacjami. Dzięki nim programista może skupić się na logice biznesowej, a nie na szczegółach technicznych związanych z manipulacją danymi w bazie.

10.1 ORM – Obiektowo-Relacyjne Mapowanie

ORM (Object-Relational Mapping) to wzorzec projektowy, który umożliwia odwzorowanie danych z relacyjnych baz danych na obiekty w kodzie aplikacji. Głównym celem ORM jest uproszczenie interakcji z bazą danych, eliminując potrzebę pisania skomplikowanych zapytań SQL. ORM pozwala na mapowanie

tabel i kolumn na klasy i właściwości obiektów, co czyni zarządzanie danymi w aplikacji bardziej intuicyjnym i zgodnym z paradygmatem programowania obiektowego. W świecie PHP popularne są dwa podejścia do ORM: Doctrine oraz Active Record.

10.1.1 Doctrine a Active Record – krótkie porównanie

Doctrine to bardziej złożone i elastyczne narzędzie ORM, które wprowadza wzorzec Unit of Work i silne rozdzielenie warstwy danych od logiki biznesowej. Dzięki Doctrine mamy pełną kontrolę nad procesem mapowania i możemy precyzyjnie definiować typy danych, relacje pomiędzy encjami oraz operacje wykonywane na danych. Doctrine pozwala na bardzo skomplikowane operacje złożone, ale wymaga od programisty nieco więcej wiedzy i konfiguracji.

Z kolei Active Record jest prostsze i bardziej bezpośrednie w użyciu. Wzorzec ten integruje logikę dostępu do danych bezpośrednio z modelem. Dzięki temu programista może bezpośrednio na instancjach modelu wykonywać operacje CRUD (Create, Read, Update, Delete), co przyspiesza i upraszcza prace nad aplikacją. Zalety Active Record to jego intuicyjność i łatwość w użyciu, szczególnie w mniejszych projektach, gdzie elastyczność Doctrine nie jest wymagana. Wadą może być mniejsza skalowalność i ograniczona kontrola nad bardziej zaawansowanymi operacjami na danych.

10.1.2 PHP 8.4: Rewolucja dzięki Property Hooks

PHP 8.4 wprowadza mechanizm Property Hooks, który można porównać do podobnego rozwiązania znanego od dawna z języka C#. W C# mechanizm ten pozwala na łatwe śledzenie zmian we właściwościach obiektów oraz wykonywanie dodatkowej logiki przy ich modyfikacji. Wprowadzenie Property Hooks do PHP otwiera nowe możliwości w kontekście ORM, zwłaszcza w kontekście uproszczonego podejścia Active Record.

Dzięki Property Hooks możliwe staje się automatyczne śledzenie zmian w polach modelu, co umożliwia implementację wzorca Active Record z interfejsem NotifyPropertyChangesContract. Ten kontrakt pozwalałby na automatyczne informowanie ORM o wszelkich zmianach, bez potrzeby ręcznej obsługi aktuali-

zacji czy walidacji danych. To znaczące uproszczenie w zarządzaniu danymi w modelach.

Co więcej, atrybuty wprowadzone w PHP 8.0 umożliwiają definiowanie dodatkowych metadanych, takich jak typy danych czy relacje między encjami, podobnie jak w Doctrine. Dzięki temu możemy połączyć intuicyjność Active Record z bardziej zaawansowanymi możliwościami definiowania struktur danych, znanych z Doctrine. Te nowe mechanizmy PHP sprawiają, że projektowanie modeli w PHP staje się nie tylko łatwiejsze, ale i bardziej elastyczne, dostosowane zarówno do mniejszych, jak i bardziej złożonych aplikacji.

10.2 Obserwator pól

Pierwszym etapem jest utworzenie klasy obserwującej zmiany pól (klasa PropertyObserver). Jej głównym zadaniem jest utrzymywanie odwołania do abstrakcyjnej klasy Model, co obejmuje wszystkie utworzone modele. Klasa ta służy przede wszystkim do gromadzenia informacji o zmodyfikowanych polach w formie tablicy, gdzie kluczami są nazwy pól, a wartościami ich aktualne wartości. Jeśli w tej tablicy znajduje się klucz główny, oznacza to konieczność wykonania aktualizacji w bazie danych. W przeciwnym wypadku, zakłada się, że model nie został jeszcze zapisany w bazie i należy go tam wprowadzić.

```
1  <?php
2
3  namespace DJWeb\Framework\DBAL\Models;
4
5  use DJWeb\Framework\DBAL\Models\Contracts\
       NotifyPropertyChangesContract;
6
7  class PropertyObserver implements
       NotifyPropertyChangesContract
8  {
9      public function __construct(private Model $model)
10     {
11     }
12     /**
13      * @var array<string, int|string|float|null>
```

```
14          */
15      private array $changedProperties = [];
16
17      public function markPropertyAsChanged(
18          string $propertyName,
19          float|int|string|null $value,
20      ): void {
21          $this->changedProperties[$propertyName] = $value;
22      }
23
24      /**
25       * @return array<string, int|string|float|null>
26       */
27      public function getChangedProperties(): array
28      {
29          return $this->changedProperties;
30      }
31
32      public bool $is_new {
33          get => !isset($this->changedProperties[$this->
                model->primary_key_name]);
34      }
35  }
```

Listing 10.1: Klasa PropertyObserver

10.3 Abstrakcyjna klasa Model

Kolejnym krokiem jest utworzenie abstrakcyjnej klasy Model, która będzie spełniała kilka kluczowych wymagań i funkcji. Po pierwsze, każda klasa dziedzicząca od tej abstrakcji będzie musiała nadpisać getter dla pola table, co umożliwi określenie, do której tabeli w bazie danych ma zostać przypisany dany model. Wprowadzimy również funkcję, która wykorzysta nową możliwość PHP 8.4, jaką jest nadpisywanie getterów dla poszczególnych pól klasy. Funkcja ta, nazwana markPropertyAsChanged, będzie odpowiedzialna za rejestrowanie zmian w polach modelu poprzez zapisanie tych informacji do obserwatora. Każdy model będzie korzystał z tej metody w swoich setterach, aby oznaczyć, które pola

wymagają zapisania w bazie danych.

W naszej bazie danych każde pole jest przechowywane jako napis, jednak dzięki użyciu typowanych właściwości (Typed Properties) dostępnych od PHP 7.4, możemy lepiej zarządzać typami danych w naszym frameworku. Aby umożliwić konwersję typów, dodamy interfejs Castable, który będzie zawierał jedną metodę from(string), zwracającą instancję samego siebie. Pozwoli to na bezproblemowe mapowanie wartości z bazy danych na typy enum oraz dowolne inne obiekty, które będą implementować ten interfejs. Dodatkowo, out of the box zapewnimy wsparcie dla castowania dat na obiekty typu Carbon, co znacznie ułatwi pracę z danymi datami w systemie.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Contracts;

interface Castable
{
    public static function from(string $value): static;
}
```

Listing 10.2: interfejs Castable

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models;

use Carbon\Carbon;
use DJWeb\Framework\DBAL\Models\Contracts\Castable;
use DJWeb\Framework\DBAL\Models\Contracts\
   PropertyChangesContract;
use DJWeb\Framework\DBAL\Models\Decorators\EntityManager;
use DJWeb\Framework\DBAL\Models\QueryBuilders\
```

```
    ModelQueryBuilder;
use DJWeb\Framework\DBAL\Models\Relations\
    RelationDecorator;
use DJWeb\Framework\DBAL\Models\Relations\RelationFactory;

abstract class Model implements PropertyChangesContract
{
    abstract public string $table { get; }

    protected(set) string $primary_key_name = 'id';
    private(set) PropertyObserver $observer;

    public int|string $id {
        get => $this->id;
        set {
            $this->id = $value;
            $this->markPropertyAsChanged('id');
        }
    }
    /**
     * @var array<string, string>
     */
    protected array $casts = [];

    public function __construct()
    {
        $this->observer = new PropertyObserver($this);
    }

    public function markPropertyAsChanged(string
        $property_name): void
    {
        $this->observer->markPropertyAsChanged(
            $property_name,
            $this->$property_name
        );
    }

    public bool $is_new {
        get => $this->observer->is_new;
    }
```

```
50
51      public function fill(array $attributes): static
52      {
53          foreach ($attributes as $key => $value) {
54              if (! property_exists($this, $key)) {
55                  continue;
56              }
57              if (isset($this->casts[$key])) {
58                  $value = $this->castAttribute($value,
                        $this->casts[$key]);
59              }
60              $this->$key = $value;
61          }
62          return $this;
63      }
64
65      public static function getTable(): string
66      {
67          return new static()->table;
68      }
69
70      protected function castAttribute(mixed $value, string
            $type): mixed
71      {
72          return match(true) {
73              $type === 'datetime' => $value instanceof
                    Carbon ? $value : Carbon::parse($value),
74              is_subclass_of($type, Castable::class) =>
                    $type::from($value),
75              default => $value,
76          };
77      }
78  }
```

Listing 10.3: klasa Model

10.4 Rozszerzamy QueryBuilder

W celu zbudowania mechanizmu operacji CRUD, skorzystamy z kontenera DI, który zwraca odpowiedni query builder (select, create, update, delete). Klasa ModelQueryBuilder, którą stworzymy, będzie służyła jako fasada dla tych operacji. Dodamy do niej publiczne pole public readonly dla fasady oraz prywatne pole builder, które będzie jednym z czterech query builderów.

Kluczowe elementy implementacji:

- Metoda select zostanie nadpisana tak, aby domyślnie przekazywała tabelę modelu do buildera.
- Metody get i first będą odpowiedzialne za pobranie danych przy pomocy bazowego buildera, a następnie zmapowanie ich na model za pomocą funkcji hydrate i hydrateMany
- Funkcje hydrate/hydrateMany wypełnią model przy pomocy metody fill, która wykorzysta wcześniej stworzone mechanizmy castowania. Poniżej przedstawiono metodę fill klasy Model

```php
<?php

abstract class Model implements
   PropertyChangesContract
{
    public function fill(array $attributes): static
    {
        foreach ($attributes as $key => $value) {
            if (! property_exists($this, $key)) {
                continue;
            }
            if (isset($this->casts[$key])) {
                $value = $this->castAttribute($value,
                   $this->casts[$key]);
            }
            $this->$key = $value;
        }
        return $this;
```

```
17      }
18  }
```

Listing 10.4: funkcja fill w klasie Model

Natomiast implementacja omawianego QueryBuildera dla modeli wygląda następująco

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DJWeb\Framework\DBAL\Models\QueryBuilders;
6
7  use DJWeb\Framework\DBAL\Contracts\Query\
       QueryBuilderFacadeContract;
8  use DJWeb\Framework\DBAL\Models\Model;
9  use DJWeb\Framework\DBAL\Query\Builders\
       DeleteQueryBuilder;
10 use DJWeb\Framework\DBAL\Query\Builders\
       InsertQueryBuilder;
11 use DJWeb\Framework\DBAL\Query\Builders\QueryBuilder;
12 use DJWeb\Framework\DBAL\Query\Builders\
       SelectQueryBuilder;
13 use DJWeb\Framework\DBAL\Query\Builders\
       UpdateQueryBuilder;
14
15 class ModelQueryBuilder
16 {
17     public readonly QueryBuilderFacadeContract $facade
           ;
18     private SelectQueryBuilder|UpdateQueryBuilder|
           InsertQueryBuilder|DeleteQueryBuilder $builder;
19
20     public function __construct(protected(set) Model
           $model)
21     {
22         $this->facade = new QueryBuilder();
23     }
```

```
24
25      public function select($columns = ['*']): static
26      {
27          $this->builder = $this->facade->select($this->
                model->table);
28          $this->builder->select($columns);
29          return $this;
30      }
31
32      public function first(): ?Model
33      {
34          $result = $this->builder->first();
35          return $result ? $this->hydrate($result) :
                null;
36      }
37
38      public function get(): array
39      {
40          $results = $this->builder->get();
41          return $this->hydrateMany($results);
42      }
43
44      /**
45       * @param string $name
46       * @param array<int|string, mixed> $arguments
47       * @return $this
48       */
49      public function __call(string $name, array
            $arguments): static
50      {
51          $this->builder->$name(...$arguments);
52          return $this;
53      }
54
55      protected function hydrate(array $attributes):
            Model
56      {
57          return $this->model->fill($attributes);
58      }
59
60      /**
```

```
61          * @param array<int, mixed> $results
62          * @return array<int, Model>
63          */
64         protected function hydrateMany(array $results):
               array
65         {
66             return array_map(fn(array $result) => $this->
                   hydrate($result),
67                 $results);
68         }
69     }
```

Listing 10.5: klasa ModelQueryBuilder

10.5 Dodajemy zapis do bazy danych

Aby umożliwić zapis do bazy danych w zgodzie z zasadami SOLID i wykorzystując połączone podejścia wzorców Doctrine i Active Record, stworzymy mechanizm składający się z trzech klas: EntityInserter, EntityUpdater oraz EntityManager. Taki podział odpowiedzialności pozwoli na efektywne zarządzanie zapisami, zarówno przez klasę modelu, jak i przez dedykowany manager.

`EntityManager` będzie pełnił rolę fasady, która zdecyduje, czy wywołać operację insert lub update na modelu, w zależności od tego, czy model już istnieje w bazie danych.

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DJWeb\Framework\DBAL\Models\Decorators;
6
7  use DJWeb\Framework\DBAL\Models\Model;
8
9  class EntityManager
10 {
```

```
11      private readonly EntityUpdater $updater;
12      private readonly EntityInserter $inserter;
13      public function __construct(private Model $model)
14      {
15          $this->updater = new EntityUpdater($this->model);
16          $this->inserter = new EntityInserter($this->model)
                ;
17      }
18
19      public function save(): void
20      {
21          if ($this->model->observer->is_new) {
22              $this->model->id = $this->inserter->insert();
23          } else {
24              $this->updater->update();
25          }
26      }
27  }
```

Listing 10.6: klasa EntityManager

EntityInserter i EntityUpdater będą odpowiadały za odpowiednie operacje wstawiania i aktualizowania danych w bazie.

Implementacja klasy EntityInserter:

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\DBAL\Models\Decorators;
6
7   use DJWeb\Framework\DBAL\Contracts\Query\
        QueryBuilderFacadeContract;
8   use DJWeb\Framework\DBAL\Models\Model;
9   use DJWeb\Framework\DBAL\Models\PropertyObserver;
10
11  class EntityInserter
12  {
```

```
13      private QueryBuilderFacadeContract $query_builder;
14      private PropertyObserver $property_watcher;
15
16      public function __construct(
17
18          private Model $model
19      ) {
20          $this->query_builder =
21              $this->model->query_builder->facade;
22          $this->property_watcher = $this->model->observer;
23      }
24
25      public function insert(): ?string
26      {
27          $builder = $this->query_builder->insert($this->
                model->table);
28          $builder->values($this->property_watcher->
                getChangedProperties())
29              ->execute();
30
31          return $builder->getInsertId();
32      }
33  }
```

Listing 10.7: klasa EntityInserter

Implementacja klasy EntityUpdater:

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\DBAL\Models\Decorators;
6
7   use DJWeb\Framework\DBAL\Contracts\Query\
        QueryBuilderFacadeContract;
8   use DJWeb\Framework\DBAL\Models\Model;
9   use DJWeb\Framework\DBAL\Models\PropertyObserver;
10
```

```php
class EntityUpdater
{
    private QueryBuilderFacadeContract $query_builder;
    private PropertyObserver $property_watcher;
    public function __construct(

        private Model $model
    ) {
        $this->query_builder = $this->model->query_builder
            ->facade;
        $this->property_watcher = $this->model->observer;
    }

    public function update(): void
    {
        $this->query_builder->update($this->model->table)
            ->set($this->property_watcher->
                getChangedProperties());
    }
}
```

Listing 10.8: klasa EntityUpdater

Model zostanie ozdobiony za pomocą dekoratora, który w swojej metodzie save wywoła save z EntityManagera. Dzięki temu zapis będzie możliwy bezpośrednio z poziomu modelu.

```php
<?php

abstract class Model implements PropertyChangesContract
{
    private EntityManager $entity_manager;

    public function __construct()
    {
        $this->entity_manager = new EntityManager($this);
    }

    public function save(): void
```

```
13      {
14          $this->entity_manager->save();
15      }
16  }
```

Listing 10.9: Zapis modelu do bazy

10.6 Dodajemy Relacje

W celu zaimplementowania obsługi relacji w naszym frameworku, skorzystamy z czterech podstawowych relacji.

- BelongsTo
- HasMany
- HasManyThrough
- BelongsToMany

Wszystkie będą implementowane z wykorzystaniem nowoczesnych możliwości PHP, takich jak atrybuty dostępne od PHP 8. Dzięki temu zachowamy spójność w działaniu frameworka, który już wykorzystuje te mechanizmy do definiowania komend, parametrów, czy getterów pól (od PHP 8.4).

10.7 Atrybut HasMany

HasMany: Ta relacja wskazuje, że jeden rekord w tabeli (np. firma) może mieć wiele powiązanych rekordów w innej tabeli (np. postów). W przykładzie, relacja HasMany między Company a Post jest zdefiniowana za pomocą atrybutu i getter zwraca wynik metody getRelation('posts'), która pobiera wszystkie posty powiązane z firmą, używając kluczy company_id i id.

```php
<?php

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\HasMany;
use DJWeb\Framework\DBAL\Models\Model;

class Company extends Model
{
    public string $table {
        get => 'companies';
    }

    #[HasMany(Post::class, foreign_key: 'company_id',
       local_key: 'id')]
    public array $posts {
        get => $this->relations->getRelation('posts');
    }
}
```

Listing 10.10: przykład relacji HasMany

10.8 Atrybut BelongsTo

BelongsTo: Ta relacja działa odwrotnie – wskazuje, że dany rekord należy do jednego powiązanego rekordu w innej tabeli. W klasie Post, BelongsTo jest powiązaniem z Company. Getter pola company zwraca odpowiednią instancję klasy Company, pobierając dane z bazy na podstawie klucza obcego company_id.

```php
<?php

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\BelongsTo;
```

```php
use DJWeb\Framework\DBAL\Models\Model;

class Post extends Model
{
    public string $table {
        get => 'posts';
    }

    public int $company_id {
        get => $this->company_id;
        set {
            $this->company_id = $value;
            $this->markPropertyAsChanged('company_id');
        }
    }

    #[BelongsTo(Company::class, foreign_key: 'company_id',
        local_key: 'id')]
    public Company $company {
        get {
            /** @var Company $model */
            $model = $this->relations->getRelation('
                company');
            return $model;
        }
    }
}
```

Listing 10.11: przykład relacji BelongsTo

Zastosowanie getterów pól (dostępnych od PHP 8.4) umożliwi lazy loading, co oznacza, że dane powiązane zostaną pobrane dopiero w momencie dostępu do pola, które reprezentuje relację. Przykład: pobranie wszystkich postów dla konkretnej firmy odbywa się tylko wtedy, gdy odwołasz się do pola $posts w klasie Company. Relacja ta dynamicznie generuje zapytanie SELECT * FROM posts WHERE company_id = ?

10.8.1 Atrybut HasManyThrough

HasManyThrough: relacja przez model pośredni Używamy jej, gdy chcemy uzyskać dostęp do modeli, które są powiązane poprzez model pośredni.

Przykładowy kod sql obrazujący tą relację

```
CREATE TABLE projects (
                            id INT PRIMARY KEY,
                            name VARCHAR(255)
);

CREATE TABLE departments (
                               id INT PRIMARY KEY,
                               project_id INT,
                               name VARCHAR(255),
                               FOREIGN KEY (project_id)
                                  REFERENCES projects(id)
);

CREATE TABLE employees (
                             id INT PRIMARY KEY,
                             department_id INT,
                             name VARCHAR(255),
                             FOREIGN KEY (department_id)
                                REFERENCES departments(id)
);


SELECT employees.*
FROM employees
         JOIN departments ON departments.id = employees.
            department_id
WHERE departments.project_id = ?;
```

Listing 10.12: relacja has many through w sql

10.8.2 Atrybut BelongsToMany

BelongsToMany - relacja wiele do wielu Używamy jej, gdy mamy powiązanie many-to-many między dwoma modelami poprzez tabelę pivot.

Przykładowy kod sql obrazujący tą relację

```
CREATE TABLE users (
                        id INT PRIMARY KEY,
                        name VARCHAR(255)
);

CREATE TABLE roles (
                        id INT PRIMARY KEY,
                        name VARCHAR(255)
);

CREATE TABLE role_user (
                            user_id INT,
                            role_id INT,
                            FOREIGN KEY (user_id)
                               REFERENCES users(id),
                            FOREIGN KEY (role_id)
                               REFERENCES roles(id)
);

SELECT roles.*
FROM roles
         JOIN role_user ON roles.id = role_user.role_id
WHERE role_user.user_id = ?;
```

Listing 10.13: relacja belongs to many w sql

10.8.3 Abstrakcyjna klasa Relation

Aby zbudować mechanizm relacji w oparciu o abstrakcyjną klasę Relation, możemy podejść do tego w sposób, który zachowa spójność i elastyczność całego frameworka. Klasa Relation będzie odpowiedzialna za pobieranie danych z bazy w postaci tablicy. Klasy pochodne, takie jak HasMany i BelongsTo, będą musiały wymusić konkretne zachowania, jak generowanie odpowiednich zapytań SQL, ustawianie warunków where oraz zwracanie modeli.

Abstrakcyjna klasa Relation będzie odpowiedzialna za ogólne operacje związane z wykonywaniem zapytań i pobieraniem danych w formie tablicy z bazy danych. Klasy pochodne, takie jak HasMany i BelongsTo, będą musiały implementować specyficzne metody, takie jak ustawianie odpowiednich warunków where oraz przekształcanie danych w modele.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models;

use DJWeb\Framework\DBAL\Contracts\Query\
   QueryBuilderContract;
use DJWeb\Framework\DBAL\Models\Contracts\RelationContract
   ;

abstract class Relation implements RelationContract
{
    protected QueryBuilderContract $query;

    /**
     * @param Model $parent
     * @param class-string<Model> $related
     * @param string $foreign_key
     * @param string $local_key
     */
    public function __construct(
        protected Model $parent,
        protected string $related,
```

```
23          protected string $foreign_key,
24          protected string $local_key,
25      ) {
26          $this->query = $this->createQueryBuilder();
27      }
28      abstract protected function createQueryBuilder():
           QueryBuilderContract;
29      abstract public function addConstraints(): void;
30
31      /**
32       * @return array<int, Model>|Model
33       */
34      public function getResults(): array|Model {
35          return $this->query->select()->get();
36      }
37
38      abstract public function getRelated(string $property):
            array|Model;
39  }
```

Listing 10.14: klasa Relation

10.8.4 Relacja HasMany

W relacji HasMany zapytanie SQL będzie zawierało warunek where, który wyszukuje wszystkie rekordy w powiązanej tabeli, gdzie klucz obcy (foreign_key) odpowiada wartości lokalnego klucza (local_key).

```
1   <?php
2
3   namespace DJWeb\Framework\DBAL\Models\Relations;
4
5   use DJWeb\Framework\DBAL\Contracts\Query\
       QueryBuilderContract;
6   use DJWeb\Framework\DBAL\Models\Model;
7   use DJWeb\Framework\DBAL\Models\Relation;
8
```

```php
class HasMany extends Relation
{

    protected function createQueryBuilder():
       QueryBuilderContract
    {
        return $this->parent->query_builder->facade
            ->select($this->related::getTable());
    }

    public function addConstraints(): void
    {
        $this->query->where(
            $this->foreign_key,
            '=',
            $this->parent->{$this->local_key}
        );
    }

    /**
     * @param string $property
     * @return array<int, Model>|Model
     */
    public function getRelated(string $property): array|
       Model
    {
        $results = $this->getResults();
        return array_map(
            fn (array $result) => new $this->related()->
               fill($result),
            $results
        );
    }
}
```

Listing 10.15: klasa HasMany

10.8.5 Relacja BelongsTo

W relacji BelongsTo, zapytanie SQL wyszukuje rekord z nadrzędnej tabeli, gdzie wartość lokalnego klucza (local_key) odpowiada wartości klucza obcego (foreign_key) w modelu podrzędnym.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Relations;

use DJWeb\Framework\DBAL\Contracts\Query\
   QueryBuilderContract;
use DJWeb\Framework\DBAL\Models\Model;
use DJWeb\Framework\DBAL\Models\Relation;

class BelongsTo extends Relation
{
    protected function createQueryBuilder():
       QueryBuilderContract
    {
        return $this->parent->query_builder->facade->
           select(
            $this->related::getTable()
        );
    }

    public function addConstraints(): void
    {
        $this->query->where(
            $this->local_key,
            '=',
            $this->parent->{$this->foreign_key}
        );
    }

    /**
     * @param string $property
     * @return array<int, Model>|Model
```

```
32          */
33      public function getRelated(string $property): array|
            Model
34      {
35          $result = $this->getResults();
36          return new $this->related()->fill($result);
37      }
38  }
```

Listing 10.16: klasa BelongsTo

10.8.6 Relacja HasManyThrough

Relacja HasManyThrough dziedziczy po relacji HasMany nadpisując jedynie warunki relacji przy wykorzystaniu operatora JOIN

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\DBAL\Models\Relations;
6
7   use DJWeb\Framework\DBAL\Models\Model;
8   use DJWeb\Framework\DBAL\Models\Relation;
9
10  class HasManyThrough extends HasMany
11  {
12      /**
13       * @param Model $parent
14       * @param class-string<Model> $related
15       * @param class-string<Model> $through
16       * @param string $first_key
17       * @param string $second_key
18       * @param string $local_key
19       * @param string $second_local_key
20       */
21      public function __construct(
```

```
22          protected Model  $parent,
23          protected string $related,
24          protected string $through,
25          protected string $first_key,
26          protected string $second_key,
27          protected string $local_key,
28          protected string $second_local_key
29      )
30      {
31          parent::__construct($parent, $related, $first_key,
                $local_key);
32      }
33
34      public function addConstraints(): void
35      {
36          $this->query
37              ->innerJoin(
38                  table: $this->through::getTable(),
39                  first: $this->through::getTable() . '.' .
                       $this->second_local_key,
40                  operator: '=',
41                  second: $this->related::getTable() . '.' .
                        $this->second_key
42              )
43              ->where(
44                  $this->through::getTable() . '.' . $this->
                       first_key,
45                  '=',
46                  $this->parent->id
47              );
48      }
49  }
```

Listing 10.17: klasa HasManyThrough

10.8.7 Relacja BelongsToMany

Relacja `BelongsToMany` dziedziczy po relacji `HasMany` nadpisując jedynie warunki relacji przy wykorzystaniu operatora `JOIN`

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Relations;

use DJWeb\Framework\DBAL\Contracts\Query\
   SelectQueryBuilderContract;
use DJWeb\Framework\DBAL\Models\Model;
use DJWeb\Framework\DBAL\Models\Relation;

class BelongsToMany extends HasMany
{
    /**
     * @param Model $parent
     * @param class-string<Model> $related
     * @param string $pivot_table
     * @param string $foreign_pivot_key
     * @param string $related_pivot_key
     */
    public function __construct(
        protected Model  $parent,
        protected string $related,
        protected string $pivot_table,
        protected string $foreign_pivot_key,
        protected string $related_pivot_key
    )
    {
        parent::__construct($parent, $related,
           $foreign_pivot_key, 'id');
    }

    public function addConstraints(): void
    {
        $this->query
```

```
            ->innerJoin(
                table: $this->pivot_table,
                first: $this->pivot_table . '.' . $this->
                    related_pivot_key,
                operator: '=',
                second: $this->related::getTable() . '.id'
            )
            ->where(
                $this->pivot_table . '.' . $this->
                    foreign_pivot_key,
                '=',
                $this->parent->id
            );
    }
}
```

Listing 10.18: klasa BelongsToMany

10.8.8 Fabryka Relacji

Aby uprościć tworzenie relacji na podstawie atrybutów modeli, możemy skorzystać ze wzorca fabryki. Fabryka relacji będzie miała dwie statyczne metody, które będą przyjmować model oraz atrybut jako parametry i na tej podstawie dynamicznie tworzyć odpowiednie relacje (HasMany, BelongsTo itd.). Takie podejście sprawi, że proces tworzenia relacji będzie bardziej zautomatyzowany, a framework stanie się bardziej elastyczny. Fabryka będzie wykorzystywać mechanizm atrybutów PHP 8 do odczytu informacji o relacjach zapisanych w modelu.

```
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Relations;

use DJWeb\Framework\DBAL\Enums\RelationType;
use DJWeb\Framework\DBAL\Models\Attributes\BelongsTo as
```

```
    BelongsToAttribute;
use DJWeb\Framework\DBAL\Models\Attributes\BelongsToMany
    as BelongsToManyAtrribute;
use DJWeb\Framework\DBAL\Models\Attributes\HasMany as
    HasManyAttribute;
use DJWeb\Framework\DBAL\Models\Attributes\HasManyThrough
    as HasManyThroughAttribute;
use DJWeb\Framework\DBAL\Models\Contracts\RelationContract
    ;
use DJWeb\Framework\DBAL\Models\Model;

class RelationFactory
{
    public static function belongsTo(
        Model $parent,
        BelongsToAttribute $attribute,
    ): RelationContract {
        $relation = new RelationFactory()->create(
            RelationType::belongsTo,
            $parent,
            $attribute->related,
            $attribute->foreign_key,
            $attribute->local_key
        );
        $relation->addConstraints();
        return $relation;
    }

    public static function belongsToMany(
        Model $parent,
        BelongsToManyAtrribute $attribute,
    ): RelationContract {
        $relation = new BelongsToMany(
            $parent,
            $attribute->related,
            $attribute->pivot_table,
            $attribute->foreign_pivot_key,
            $attribute->related_pivot_key
        );
        $relation->addConstraints();
        return $relation;
```

```
45      }
46
47      public static function hasManyThrough(
48          Model $parent,
49          HasManyThroughAttribute $attribute,
50      ): RelationContract {
51          $relation = new HasManyThrough(
52              $parent,
53              $attribute->related,
54              $attribute->through,
55              $attribute->first_key,
56              $attribute->second_key,
57              $attribute->local_key,
58              $attribute->second_local_key
59          );
60          $relation->addConstraints();
61          return $relation;
62      }
63
64      public static function hasMany(
65          Model $parent,
66          HasManyAttribute $attribute,
67      ): RelationContract {
68          $relation = new RelationFactory()->create(
69              RelationType::hasMany,
70              $parent,
71              $attribute->related,
72              $attribute->foreign_key,
73              $attribute->local_key
74          );
75          $relation->addConstraints();
76          return $relation;
77      }
78
79      /**
80       * @param RelationType $type
81       * @param Model $parent
82       * @param class-string<Model> $related
83       * @param string $foreignKey
84       * @param string $localKey
85       *
```

```
86          * @return RelationContract
87          */
88         private function create(
89             RelationType $type,
90             Model $parent,
91             string $related,
92             string $foreignKey,
93             string $localKey
94         ): RelationContract {
95             return match ($type->value) {
96                 'hasMany' => new HasMany($parent, $related,
                       $foreignKey, $localKey),
97                 'belongsTo' => new BelongsTo(
98                     $parent,
99                     $related,
100                    $foreignKey,
101                    $localKey
102                ),
103            };
104    }
105 }
```

Listing 10.19: klasa RelationFactory

Fabryka relacji umożliwia jednolity i elastyczny sposób tworzenia relacji, niezależnie od ich rodzaju. Dzięki temu framework może łatwo obsługiwać zarówno HasMany, BelongsTo, jak i inne typy relacji. Dzięki atrybutom i fabryce, proces tworzenia relacji jest zautomatyzowany. Użytkownik musi jedynie zadeklarować atrybuty w modelach, a resztę obsługuje mechanizm frameworka.

10.8.9 Dekorator Relacji

Mechanizm obsługi relacji w naszym frameworku opiera się na dekoratorze klasy Model, który zarządza relacjami za pomocą dwóch prywatnych tablic pełniących rolę uproszczonego cache. Jedna z tablic przechowuje informacje o zdeklarowanych relacjach, a druga o wartościach już pobranych z bazy danych. Dzięki temu mechanizmowi, relacje są tworzone i zarządzane dynamicznie, bez

konieczności wielokrotnego przetwarzania tych samych danych. Kiedy w klasie pochodnej wywoływana jest metoda getRelation, framework parsuje atrybuty, a następnie przy pomocy fabryki relacji generuje odpowiednie zapytanie, pobiera dane i zapisuje je w cache, co znacząco zwiększa wydajność.

Kluczową rolę w tym mechanizmie odgrywa asynchroniczna widzialność pól, dostępna od PHP 8.4. Dzięki niej, pole relations, ma widoczność ustawioną na protected private(set), co oznacza, że jest oznaczone jako readonly w klasach pochodnych, co zapewnia bezpieczny dostęp do relacji, jednak w klasie bazowej możemy je ustawiać w dowolnym momencie, na przykład podczas ładowania relacji z bazy. To podejście eliminuje konieczność tworzenia dodatkowych getterów czy mapowania metod, zachowując prostotę i czytelność kodu, a jednocześnie zapewnia pełną kontrolę nad mechanizmem relacji w modelach.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Relations;

use DJWeb\Framework\DBAL\Models\Attributes\BelongsTo as
    BelongsToAttribute;
use DJWeb\Framework\DBAL\Models\Attributes\BelongsToMany
    as BelongsToManyAttribute;
use DJWeb\Framework\DBAL\Models\Attributes\HasMany as
    HasManyAttribute;
use DJWeb\Framework\DBAL\Models\Attributes\HasManyThrough
    as HasManyThroughAttribute;
use DJWeb\Framework\DBAL\Models\Contracts\RelationContract
    ;
use DJWeb\Framework\DBAL\Models\Model;
use ReflectionAttribute;
use ReflectionProperty;

class RelationDecorator
{
    /**
     * @var array<string, RelationContract|null>
     */
    private array $relations = [];
```

```
22
23      /**
24       * @var array<string, Model|array<int|string, mixed>>
25       */
26      private array $relationsCache = [];
27
28      public function __construct(private readonly Model
            $model)
29      {
30      }
31
32      /**
33       * @param string $name
34       *
35       * @return Model|array<int|string, Model>|null
36       *
37       * @throws \ReflectionException
38       */
39      public function getRelation(string $name): Model|array
            |null
40      {
41          if (! isset($this->relations[$name])) {
42              $property = new ReflectionProperty($this->
                    model, $name);
43              $this->initializeRelation($property);
44          }
45          $exception = new \RuntimeException("Relation {
                $name} not found");
46          $this->relationsCache[$name]
47              ??= $this->relations[$name]?->getRelated() ??
                        throw $exception;
48          return $this->relationsCache[$name];
49      }
50
51      private function initializeRelation(ReflectionProperty
            $property): void
52      {
53          $this->initializeAllRelations(
54              $property,
55              BelongsToAttribute::class,
56              $this->initializeBelongsTo(...)
```

```
57          );
58          $this->initializeAllRelations(
59              $property,
60              HasManyAttribute::class,
61              $this->initializeHasMany(...)
62          );
63          $this->initializeAllRelations(
64              $property,
65              BelongsToManyAttribute::class,
66              $this->initializeBelongsToMany(...)
67          );
68          $this->initializeAllRelations(
69              $property,
70              HasManyThroughAttribute::class,
71              $this->initializeHasManyThrough(...)
72          );
73      }
74
75      private function initializeAllRelations(
76          ReflectionProperty $property,
77          string $type,
78          callable $callback
79      ): void {
80          $attributes = $property->getAttributes();
81          $attributes = array_filter(
82              $attributes,
83              static fn ($attribute) => $attribute->getName
                    () === $type
84          );
85          array_walk(
86              $attributes,
87              static fn (ReflectionAttribute $attribute) =>
                    $callback(
88                   $property,
89                   $attribute->newInstance()
90              )
91          );
92      }
93
94      private function initializeBelongsToMany(
95          ReflectionProperty $property,
```

```
96          BelongsToManyAttribute $attribute
97      ): void {
98          $value = RelationFactory::belongsToMany($this->
                model, $attribute);
99          $this->relations[$property->getName()] = $value;
100     }
101
102     private function initializeBelongsTo(
103         ReflectionProperty $property,
104         BelongsToAttribute $attribute
105     ): void {
106         $value = RelationFactory::belongsTo($this->model,
                $attribute);
107         $this->relations[$property->getName()] = $value;
108     }
109
110     private function initializeHasManyThrough(
111         ReflectionProperty $property,
112         HasManyThroughAttribute $attribute
113     ): void {
114         $value = RelationFactory::hasManyThrough($this->
                model, $attribute);
115         $this->relations[$property->getName()] = $value;
116     }
117
118     private function initializeHasMany(
119         ReflectionProperty $property,
120         HasManyAttribute $attribute
121     ): void {
122         $value = RelationFactory::hasMany($this->model,
                $attribute);
123         $this->relations[$property->getName()] = $value;
124     }
125 }
```

Listing 10.20: klasa RelationFactory

10.9 Dodajemy Fabrykę

W środowiskach testowych kluczowe jest szybkie wypełnianie bazy danych odpowiednimi danymi testowymi. Zamiast tworzyć mechanizm generowania danych od zera, skorzystamy z dojrzałej i popularnej biblioteki Faker/Faker, która oferuje metody generowania różnych typów danych w wielu językach. Dzięki niej możemy w prosty sposób generować fikcyjne dane takie jak imiona, nazwiska, adresy, numery telefonów i wiele innych, co znacząco ułatwia proces przygotowywania testów. Aby ją dodać do projektu należy wykonać $composer require fakerphp/faker$

W ramach naszego rozwiązania stworzymy bazową fabrykę modeli, która będzie przyjmować tablicę atrybutów i automatycznie wypełniać model za pomocą jego metody fill. Umożliwi to szybkie tworzenie instancji modeli z danymi testowymi. Aby jeszcze bardziej uprościć ten proces, dodamy komendę MakeFactory, która pozwoli w łatwy sposób generować fabryki dla różnych modeli, umożliwiając szybkie i efektywne przygotowanie danych testowych w całym projekcie.

10.9.1 Mechanizm działania

Abstrakcyjna klasa Factory będzie podstawą dla fabryk modeli, a jej implementacja jest prosta i efektywna. W konstruktorze klasa ta przyjmuje instancję Generator z biblioteki Faker, która umożliwia generowanie danych testowych. Klasa udostępni kilka metod: make, create, oraz createMany, które automatyzują proces tworzenia i wypełniania modeli danymi testowymi.

Główne metody klasy Factory:

- make: Tworzy instancję modelu i wypełnia go wygenerowanymi danymi, ale nie zapisuje go do bazy danych. Dzięki temu można pracować z modelem bez od razu zapisania go w bazie.
- create: Tworzy model, wypełnia go danymi, a następnie zapisuje go do bazy danych.
- createMany: Tworzy wiele instancji modeli w pętli, korzystając z instrukcji językowej range do określenia liczby instancji, a następnie zapisuje każdy model do bazy danych.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models;

use Faker\Factory as FakerFactory;
use Faker\Generator;

abstract class Factory
{
    public function __construct(protected ?Generator
        $faker = null)
    {
        $this->faker ??= FakerFactory::create();
    }

    public function make(array $attributes = []): Model
    {
        $modelClass = $this->getModelClass();
        $model = new $modelClass();
        $data = [...$this->definition(), ...$attributes];
        $model->fill($data);
        return $model;
    }

    public function create(array $attributes = []): Model
    {
        $model = $this->make($attributes);
        $model->save();
        return $model;
    }

    /**
     * @param int $count
     * @param array<int, Model> $attributes
     * @return array
     */
    public function createMany(int $count, array
        $attributes = []): array
    {
```

```
40          $models = [];
41          foreach (range(1, $count) as $i) {
42              $models []= $this->create($attributes);
43          }
44          return $models;
45      }
46
47      abstract public function definition(): array;
48      /**
49       * @return class-string<Model>
50       */
51      abstract protected function getModelClass(): string;
52  }
```

Listing 10.21: klasa Factory

10.9.2 Komenda MakeFactory

Mimo że współczesne PHP oferuje rozbudowane mechanizmy typowania danych, stworzenie klasy automatycznie generującej fabrykę na podstawie modelu nie musi być skomplikowane. Możemy to osiągnąć, wykorzystując mechanizmy szablonów (stubów), enumów oraz atrybutów, które znacznie upraszczają proces definiowania atrybutów modelu i ich powiązania z odpowiednimi metodami generatora danych.

```
1   <?php
2
3   namespace DJWeb\Framework\Enums;
4
5   enum FakerMethod: string
6   {
7       case NAME = 'name';
8       case EMAIL = 'safeEmail';
9       case PHONE = 'phoneNumber';
10      case DATE = 'date';
11      case TIME = 'time';
```

```
12      case ADDRESS = 'address';
13      case COMPANY = 'company';
14      case CITY = 'city';
15      case STATE = 'state';
16      case COUNTRY = 'country';
17      case PASSWORD = 'password';
18  }
```

Listing 10.22: enum FakerMethod

```
1   <?php
2
3   namespace DJWeb\Framework\DBAL\Models\Attributes;
4
5   use Attribute;
6   use DJWeb\Framework\Enums\FakerMethod;
7
8   #[Attribute(Attribute::TARGET_PROPERTY)]
9   readonly class FakeAs
10  {
11      public function __construct(public FakerMethod $method
            )
12      {}
13  }
```

Listing 10.23: atrybut FakeAs

Dzięki podejściu opartemu o SOLID możemy stworzyć o klasę złożoności poniżej 5, która automatycznie generuje fabryki na podstawie modelu, zachowując przejrzystość i prostotę implementacji. Wykorzystanie enumów pozwoli zdefiniować powiązanie między polami modelu a metodami generatora Faker, co automatyzuje proces wypełniania danych.

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

```
4
5  namespace DJWeb\Framework\Console\Commands;
6
7  use DJWeb\Framework\Console\Attributes\AsCommand;
8  use DJWeb\Framework\Container\Contracts\ContainerContract;
9  use DJWeb\Framework\DBAL\Models\Attributes\FakeAs;
10 use ReflectionClass;
11 use ReflectionProperty;
12
13 #[AsCommand(name: 'make:factory')]
14 class MakeFactory extends MakeCommand
15 {
16     protected $fakerClass;
17
18     public function __construct(ContainerContract
       $container)
19     {
20         parent::__construct($container);
21         $class = $container->getBinding('app.faker_class')
               ?? FakeAs::class;
22         $this->fakerClass = $class;
23     }
24     protected function getStub(): string
25     {
26         $dir = dirname(__DIR__, 3);
27
28         return $dir . '/stubs/factory.stub';
29     }
30
31     protected function getDefaultNamespace(): string
32     {
33         return $this->rootNamespace() . 'Database\\
           Factories';
34     }
35
36     protected function buildClass(string $name): string
37     {
38         $modelClass = $this->getModelClass($name);
39
40         $stub = parent::buildClass($name);
41
```

```
42          $stub = str_replace('DummyModel', $modelClass,
                $stub);
43
44          $reflection = new ReflectionClass($modelClass);
45          $properties = $reflection->getProperties(
                ReflectionProperty::IS_PUBLIC);
46          $properties = array_filter(
47              $properties,
48              fn (ReflectionProperty $property) => !!
                    $property->getAttributes($this->fakerClass)
49          );
50
51          $definitionContent = '';
52          foreach ($properties as $property) {
53              $propertyName = $property->getName();
54              $fakerMethod = $this->guessFakerMethod(
                    $property);
55              $definitionContent .= "            '{
                    $propertyName}' => \$this->faker->{
                    $fakerMethod}(),\n";
56          }
57
58          return str_replace('// DummyDefinition',
                $definitionContent, $stub);
59      }
60
61      public function getModelClass(string $name)
62      {
63          $modelClass = str_replace('Factory', '', $name);
64          $modelClass = str_replace('.php', '', $modelClass)
                ;
65          return '\\' . $this->rootNamespace() . 'Database\\
                Models\\' . $modelClass;
66      }
67
68      private function guessFakerMethod(ReflectionProperty
            $property): string
69      {
70          $attributes = $property->getAttributes(FakeAs::
                class);
71          $attribute = $attributes[0];
```

```
72          /** @var FakeAs $fake */
73          $fake = $attribute->newInstance();
74          return $fake->method->value;
75      }
76
77      protected function getPath(string $name): string
78      {
79          $name = str_replace('\\', '/', $name);
80
81          return $this->container->getBinding(
82                  'app.factories_path'
83              ) . '/' . $name;
84      }
85  }
```

Listing 10.24: komenda MakeFactory

Szablon do generowania fabryk wygląda następująco:

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DummyRootNamespace;
6
7   use DJWeb\Framework\DBAL\Models\Factory;
8
9   class DummyClass extends Factory
10  {
11      protected function getModelClass(): string
12      {
13          return DummyModel::class;
14      }
15
16      public function definition(): array
17      {
18          return [
19  // DummyDefinition
20          ];
```

```
21      }
22  };
```

Listing 10.25: szablon do fabryk

10.10 Dodajemy Seedery

Na koniec dodamy mechanizm seederów. Jest to doskonałe dopełnienie systemu zarządzania bazą danych, ponieważ umożliwia szybkie i łatwe wypełnienie bazy danymi testowymi lub początkowymi. Składa się on z trzech prostych elementów: abstrakcyjnej klasy Seeder, która definiuje strukturę seederów, oraz dwóch komend: MakeSeeder i DatabaseSeeder.

10.10.1 Mechanizm działania

Abstrakcyjna klasa Seeder posiada jedną abstrakcyjną metodę run, którą każda klasa dziedzicząca musi zaimplementować. To właśnie w tej metodzie definiujemy logikę dodawania danych do bazy. Metoda call umożliwia uruchomienie innego seedera, co pozwala na zagnieżdżanie seederów i uruchamianie ich w odpowiedniej kolejności.

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DJWeb\Framework\DBAL\Models;
6
7  abstract class Seeder
8  {
9      abstract public function run(): void;
10
11     protected function call(string $seeder): void
12     {
```

```
13          $instance = new $seeder();
14          $instance->run();
15      }
16  }
```

Listing 10.26: klasa Seeder

10.10.2 Komenda MakeSeeder

Komenda MakeSeeder odpowiada za generowanie plików seederów na podstawie przygotowanych szablonów (stubów). Pozwala na szybkie tworzenie nowych seederów bez konieczności ręcznego pisania pliku od podstaw.

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\Console\Commands;
6
7   use DJWeb\Framework\Console\Attributes\AsCommand;
8
9   #[AsCommand(name: 'make:seeder')]
10  class MakeSeeder extends MakeCommand
11  {
12      protected function getStub(): string
13      {
14          $dir = dirname(__DIR__, 3);
15
16          return $dir . '/stubs/seeder.stub';
17      }
18
19      protected function getDefaultNamespace(): string
20      {
21          return $this->rootNamespace() . 'Database\\Seeders
                ';
22      }
23
```

```
24      protected function getPath(string $name): string
25      {
26          $name = str_replace('\\', '/', $name);
27
28          return $this->container->getBinding(
29                  'app.seeders_path'
30              ) . '/' . $name;
31      }
32  }
```

Listing 10.27: Komenda MakeSeeder

10.10.3 Komenda DatabaseSeed

Komenda DatabaseSeeder odpowiada za uruchomienie funkcji run dla określonej klasy seedera podanej jako parametr. Dzięki temu można w prosty sposób wypełniać bazę danych za pomocą konkretnego seederu.

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\Console\Commands;
6
7   use DJWeb\Framework\Console\Attributes\AsCommand;
8   use DJWeb\Framework\Console\Attributes\CommandArgument;
9   use DJWeb\Framework\Console\Command;
10
11  #[AsCommand(name: 'database:seed')]
12  class DatabaseSeed extends Command
13  {
14      #[CommandArgument(name: 'seeder', value: '
            DatabaseSeeder', description: 'The class name of
            the root seeder')]
15      protected string $seeder = 'DatabaseSeeder';
16
17      public function run(): int
```

```
18      {
19          $seederClass = $this->rootNamespace() . 'Database
                \\Seeders\\' . $this->seeder;
20
21          if (!class_exists($seederClass)) {
22              $this->getOutput()->error("Seeder class {
                    $seederClass} does not exist.");
23              return 1;
24          }
25
26          $seeder = new $seederClass();
27          $seeder->run();
28
29          $this->getOutput()->info('Database seeding
                completed successfully.');
30          return 0;
31      }
32
33      protected function rootNamespace(): string
34      {
35          return $this->container->getBinding('app.
                root_namespace');
36      }
37  }
```

Listing 10.28: Komenda DatabaseSeed

10.11 Weryfikacja poprawności działania

W tej części rozdziału pokażemy kilka przykładowych testów dla obsługi modeli

10.11.1 Przykładowe modele

```php
<?php

declare(strict_types=1);

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Model;

class Comment extends Model
{

    public string $table {
        get => 'comments';
    }

    public string $content {
        get => $this->content;
        set {
            $this->content = $value;
            $this->markPropertyAsChanged('content');
        }
    }

    public string $post_id {
        get => $this->post_id;
        set {
            $this->post_id = $value;
            $this->markPropertyAsChanged('post_id');
        }
    }
}
```

Listing 10.29: model Comment

```php
<?php

namespace Tests\Helpers\Models;

```

```php
use DJWeb\Framework\DBAL\Models\Attributes\HasMany;
use DJWeb\Framework\DBAL\Models\Attributes\HasManyThrough;
use DJWeb\Framework\DBAL\Models\Model;

class Company extends Model
{
    public string $table {
        get => 'companies';
    }

    #[HasMany(Post::class, foreign_key: 'company_id',
       local_key: 'id')]
    public array $posts {
        get => $this->relations->getRelation('posts');
    }

    #[HasManyThrough(Comment::class, Post::class, '
       company_id', 'post_id', 'id', 'id')]
    public array $comments {
        get => $this->relations->getRelation('comments');
    }
}
```

Listing 10.30: model Company

```php
<?php

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Model;

class CompanyUser extends Model
{

    public string $table {
        get => 'company_users';
    }

    public int $company_id {
```

```
        get => $this->company_id;
        set {
            $this->company_id = $value;
            $this->markPropertyAsChanged('company_id');
        }
    }

    public int $user_id {
        get => $this->user_id;
        set {
            $this->user_id = $value;
            $this->markPropertyAsChanged('user_id');
        }
    }
}
```

Listing 10.31: model Company User

```
<?php

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\BelongsTo;
use DJWeb\Framework\DBAL\Models\Attributes\FakeAs;
use DJWeb\Framework\DBAL\Models\Model;
use DJWeb\Framework\Enums\FakerMethod;
use Tests\Helpers\Casts\Status;

class Post extends Model
{
    public string $table {
        get => 'posts';
    }

    public Status $status {
        get => $this->status;
        set {
            $this->status = $value;
            $this->markPropertyAsChanged('status');
```

```
        }
    }
    #[FakeAs(FakerMethod::NAME)]
    public string $name {
        get => $this->name;
        set {
            $this->name = $value;
            $this->markPropertyAsChanged('name');
        }
    }

    public int $company_id {
        get => $this->company_id;
        set {
            $this->company_id = $value;
            $this->markPropertyAsChanged('company_id');
        }
    }

    #[BelongsTo(Company::class, foreign_key: 'company_id',
        local_key: 'id')]
    public Company $company {
        get {
            /** @var Company $model */
            $model = $this->relations->getRelation('
               company');
            return $model;
        }
    }

    protected array $casts = [
        'published_at' => 'datetime',
        'status' => Status::class,
    ];
}
```

Listing 10.32: model Post

```
<?php
```

```php

declare(strict_types=1);

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\BelongsToMany;
use DJWeb\Framework\DBAL\Models\Model;

class User extends Model
{

    public string $table {
        get => 'users';
    }

    public string $name {
        get => $this->name;
        set {
            $this->name = $value;
            $this->markPropertyAsChanged('name');
        }
    }

    #[BelongsToMany(Company::class, 'user_company', '
       user_id', 'company_id')]
    public array $companies {
        get => $this->relations->getRelation('companies');
    }
}
```

Listing 10.33: model User

10.11.2 Podstawowe testy modelu

W tej części sprawdzimy poprawnośc podstawowe operacja wykonywane na modelach

```php
<?php

declare(strict_types=1);

namespace Tests\Models;

use DJWeb\Framework\Base\Application;
use DJWeb\Framework\DBAL\Contracts\ConnectionContract;
use DJWeb\Framework\DBAL\Contracts\Query\
   DeleteQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
   InsertQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
   SelectQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
   UpdateQueryBuilderContract;
use DJWeb\Framework\DBAL\Query\Builders\DeleteQueryBuilder
   ;
use DJWeb\Framework\DBAL\Query\Builders\InsertQueryBuilder
   ;
use DJWeb\Framework\DBAL\Query\Builders\QueryBuilder;
use DJWeb\Framework\DBAL\Query\Builders\SelectQueryBuilder
   ;
use DJWeb\Framework\DBAL\Query\Builders\UpdateQueryBuilder
   ;
use Tests\BaseTestCase;
use Tests\Helpers\Casts\Status;
use Tests\Helpers\Models\Post;

class ModelTest extends BaseTestCase
{
    private QueryBuilder $queryBuilder;
    private ConnectionContract $mockConnection;

    public function testHydrateStatus()
    {
        $post = new Post();
        $post->fill([
            'status' => 'published',
        ]);
        $this->assertEquals(
```

```
34              Status::published,
35              $post->status
36          );
37      }
38
39      public function testIsNew()
40      {
41          $post = new Post();
42          $this->assertTrue($post->is_new);
43      }
44
45      public function testAfterInsertModelIsNotLongerNew()
46      {
47          $post = new Post();
48          $post->status = Status::published;
49          $mockPDOStatement = $this->createMock(\
                PDOStatement::class);
50          $this->mockConnection->expects($this->once())
51              ->method('query')
52              ->willReturn($mockPDOStatement);
53          $this->mockConnection->expects($this->once())
54              ->method('getLastInsertId')
55              ->willReturn('1');
56          $post->save();
57          $this->assertFalse($post->is_new);
58          $this->assertEquals('1', $post->id);
59      }
60
61      public function testFirst()
62      {
63
64          $mockPDOStatement = $this->createMock(\
                PDOStatement::class);
65          $mockPDOStatement->expects($this->once())
66              ->method('fetchAll')
67              ->with(\PDO::FETCH_ASSOC)
68              ->willReturn([
69                  [
70                      'id' => 1,
71                      'status' => 'published',
72                      'created_at' => '2024-01-01 00:00:00',
```

```
                ]
            ]);
        $this->mockConnection->expects($this->once())
            ->method('query')
            ->willReturn($mockPDOStatement);

        $query = Post::query();
        $post = $query->select()->where('id', '=', '1')->
            first();
        $this->assertInstanceOf(Post::class, $post);
    }

    public function testUpdate()
    {

        $mockPDOStatement = $this->createMock(\
            PDOStatement::class);
        $mockPDOStatement->expects($this->once())
            ->method('fetchAll')
            ->with(\PDO::FETCH_ASSOC)
            ->willReturn([
                [
                    'id' => 1,
                    'status' => 'published',
                    'created_at' => '2024-01-01 00:00:00',
                ]
            ]);
        $this->mockConnection->expects($this->once())
            ->method('query')
            ->willReturn($mockPDOStatement);

        $query = Post::query();
        $post = $query->select()->where('id', '=', '1')->
            first();
        $post->status = Status::draft;
        $post->save();
    }

    public function testGet()
    {

```

```
111         $mockPDOStatement = $this->createMock(\
                PDOStatement::class);
112         $mockPDOStatement->expects($this->once())
113             ->method('fetchAll')
114             ->with(\PDO::FETCH_ASSOC)
115             ->willReturn([
116                 [
117                     'id' => 1,
118                     'status' => 'published',
119                     'created_at' => '2024-01-01 00:00:00',
120                 ]
121             ]);
122         $this->mockConnection->expects($this->once())
123             ->method('query')
124             ->willReturn($mockPDOStatement);
125 
126         $query = Post::query();
127         $posts = $query->select()->where('id', '=', '1')->
                get();
128         $this->assertIsArray($posts);
129         $this->assertInstanceOf(Post::class, $posts[0]);
130     }
131 
132     protected function setUp(): void
133     {
134         $this->mockConnection = $this->createMock(
                ConnectionContract::class);
135         Application::getInstance()->set(
136             InsertQueryBuilderContract::class,
137             new InsertQueryBuilder($this->mockConnection)
138         );
139         Application::getInstance()->set(
140             UpdateQueryBuilderContract::class,
141             new UpdateQueryBuilder($this->mockConnection)
142         );
143         Application::getInstance()->set(
144             DeleteQueryBuilderContract::class,
145             new DeleteQueryBuilder($this->mockConnection)
146         );
147         Application::getInstance()->set(
148             SelectQueryBuilderContract::class,
```

```
149                 new SelectQueryBuilder($this->mockConnection)
150             );
151             $this->queryBuilder = new QueryBuilder();
152         }
153 }
```

Listing 10.34: podstawowe testy modelu

10.11.3 Testy fabryki

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace Tests\Models;
6
7  use DJWeb\Framework\Base\Application;
8  use DJWeb\Framework\DBAL\Contracts\ConnectionContract;
9  use DJWeb\Framework\DBAL\Contracts\Query\
       InsertQueryBuilderContract;
10 use DJWeb\Framework\DBAL\Models\Factory;
11 use PDOStatement;
12 use PHPUnit\Framework\TestCase;
13 use Tests\BaseTestCase;
14 use Tests\Helpers\Casts\Status;
15 use Tests\Helpers\Models\Post;
16
17 class FactoryTest extends BaseTestCase
18 {
19     public function testFactory()
20     {
21         $class = new class extends Factory
22         {
23
24             public function definition(): array
25             {
26                 return [
```

```
27                      'name' => 'test',
28                      'status' => 'draft',
29                  ];
30              }
31
32              protected function getModelClass(): string
33              {
34                  return Post::class;
35              }
36          };
37
38          $app = Application::getInstance();
39          $builder = $this->createMock(
                InsertQueryBuilderContract::class);
40          $any = $this->any();
41          $stmt = $this->createMock(PDOStatement::class);
42          $builder->expects($any)->method('table')->
                willReturnSelf();
43          $builder->expects($any)->method('values')->
                willReturnSelf();
44          $builder->expects($any)->method('execute')->
                willReturn($stmt);
45          $builder->expects($any)->method('getInsertId')->
                willReturn('1');
46          $app->set(InsertQueryBuilderContract::class,
                $builder);
47
48          $factory = new $class();
49          $post = $factory->create();
50          $this->assertInstanceOf(Post::class, $post);
51          $this->assertEquals(Status::draft, $post->status);
52
53          $posts = $factory->createMany(2);
54          $this->assertCount(2, $posts);
55          $this->assertEquals(Status::draft, $posts[0]->
                status);
56          $this->assertEquals(Status::draft, $posts[1]->
                status);
57      }
58  }
```

Listing 10.35: testy fabryki

10.11.4 Testy relacji

Poniżej przedstawiono test dla relacji `HasMany` testy pozostałych relacji są analogiczne

```php
<?php

declare(strict_types=1);

namespace Models;

use DJWeb\Framework\Base\Application;
use DJWeb\Framework\DBAL\Contracts\ConnectionContract;
use DJWeb\Framework\DBAL\Contracts\Query\
    DeleteQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
    InsertQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
    SelectQueryBuilderContract;
use DJWeb\Framework\DBAL\Contracts\Query\
    UpdateQueryBuilderContract;
use DJWeb\Framework\DBAL\Query\Builders\DeleteQueryBuilder
    ;
use DJWeb\Framework\DBAL\Query\Builders\InsertQueryBuilder
    ;
use DJWeb\Framework\DBAL\Query\Builders\QueryBuilder;
use DJWeb\Framework\DBAL\Query\Builders\SelectQueryBuilder
    ;
use DJWeb\Framework\DBAL\Query\Builders\UpdateQueryBuilder
    ;
use Tests\BaseTestCase;
use Tests\Helpers\Models\Company;
use Tests\Helpers\Models\Post;

class HasManyTest extends BaseTestCase
{
    private QueryBuilder $queryBuilder;
    private ConnectionContract $mockConnection;

    public function testHasMany()
```

```
28      {
29          $mockPDOStatement = $this->createMock(\
                PDOStatement::class);
30          $mockPDOStatement->expects($this->once())
31              ->method('fetchAll')
32              ->with(\PDO::FETCH_ASSOC)
33              ->willReturn([
34                  [
35                      'id' => 1,
36                      'created_at' => '2024-01-01 00:00:00',
37                  ]
38              ]);
39          $this->mockConnection->expects($this->once())
40              ->method('query')
41              ->willReturn($mockPDOStatement);
42          $company = new Company();
43          $company->id = 1;
44          $this->assertIsArray($company->posts);
45
46          $this->assertInstanceOf(Post::class, $company->
                posts[0]);
47      }
48
49      protected function setUp(): void
50      {
51          $this->mockConnection = $this->createMock(
                ConnectionContract::class);
52          Application::getInstance()->set(
53              InsertQueryBuilderContract::class,
54              new InsertQueryBuilder($this->mockConnection)
55          );
56          Application::getInstance()->set(
57              UpdateQueryBuilderContract::class,
58              new UpdateQueryBuilder($this->mockConnection)
59          );
60          Application::getInstance()->set(
61              DeleteQueryBuilderContract::class,
62              new DeleteQueryBuilder($this->mockConnection)
63          );
64          Application::getInstance()->set(
65              SelectQueryBuilderContract::class,
```

```
66              new SelectQueryBuilder($this->mockConnection)
67          );
68          $this->queryBuilder = new QueryBuilder();
69      }
70  }
```

Listing 10.36: testy relacji HasMany

10.11.5 Testy seedera

```
1   <?php
2
3   namespace Tests\Console\Commands;
4
5   use DJWeb\Framework\Console\Application;
6   use DJWeb\Framework\Console\Output\Contacts\OutputContract
        ;
7   use DJWeb\Framework\Container\Contracts\ContainerContract;
8   use Tests\BaseTestCase;
9
10  class MakeSeederTest extends BaseTestCase
11  {
12      private Application $app;
13      private OutputContract $output;
14      public function testMakeSeeder()
15      {
16          $seederName = 'UserSeeder';
17          $file = 'UserSeeder.php';
18          $_SERVER['argv'] = ['console/bin', 'make:seeder',
                $seederName];
19
20          $this->app->set(OutputContract::class, $this->
                output);
21
22          $this->output->expects($this->once())
23              ->method('info')
24              ->with("Utworzono {$file}");
```

```
25
26          $result = $this->app->handle();
27
28          $this->assertEquals(0, $result);
29          $this->assertFileExists(sys_get_temp_dir() . '/' .
                $file);
30
31      }
32      public function setUp(): void
33      {
34          Application::withInstance(null);
35          $this->app = Application::getInstance();
36          $this->app->bind(
37              'app.base_path',
38              sys_get_temp_dir()
39          );
40          $this->app->bind('app.seeders_path',
               sys_get_temp_dir());
41          $this->output = $this->createMock(OutputContract::
               class);
42          $this->app->set(ContainerContract::class, $this->
               app);
43
44      }
45  }
```

Listing 10.37: testy seedera

10.11.6 Testy manualne

Powtórzymy, tym razem jednak, ją wykonując procedurę testową przedstawioną na początku rozdziału 7.

Ważne: Docelowy model i migracja dla tabeli users zostaną pokazane w dalszej części książki w rodziale dotyczącym autoryzacji

Stworzenie migracji dla tabeli użytkowników

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

następnie musimy uzupełnić tą migrację zgodnie z zaplanowaną strukturą bazy danych

```php
<?php

declare(strict_types=1);

namespace App\Database\Migrations;

use DJWeb\Framework\DBAL\Migrations\Migration;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
    DateTimeColumn;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\IntColumn;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
    PrimaryColumn;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
    VarcharColumn;

return new class extends Migration
{
    /**
     * run migration.
     */
    public function up(): void
    {
        $this->schema->createTable('users', [
            new IntColumn('id', nullable: false,
                autoIncrement: true),
            new VarcharColumn('name'),
            new VarcharColumn('email'),
            new VarcharColumn('password'),
            new DateTimeColumn('created_at', current: true
                ),
            new DateTimeColumn('updated_at',
                currentOnUpdate: true),
            new PrimaryColumn('id'),
        ]);
        $this->schema->uniqueIndex('users', '
```

```
            unique_users_email', 'email');
    }

    /**
     * rollback migration.
     */
    public function down(): void
    {
        $this->schema->dropTable('users');
    }
};
```

Listing 10.38: migracja tabeli users

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
```

Należy pamiętać aby wygenerowany model dziedziczył po naszym bazowym modelu User co zapewni automatyczne hashowanie haseł.

```
<?php

declare(strict_types=1);

namespace App\Database\Models;

use Carbon\Carbon;
use DJWeb\Framework\DBAL\Models\Attributes\FakeAs;
use DJWeb\Framework\DBAL\Models\Entities\User as BaseUser;
use DJWeb\Framework\Enums\FakerMethod;

class User extends BaseUser
{
    public string $table {
        get => 'users';
```

```
16      }
17
18      #[FakeAs(FakerMethod::NAME)]
19      public string $name {
20          get => $this->name;
21          set {
22              $this->name = $value;
23              $this->markPropertyAsChanged('name');
24          }
25      }
26      #[FakeAs(FakerMethod::EMAIL)]
27      public string $email {
28          get => $this->email;
29          set {
30              $this->email = $value;
31              $this->markPropertyAsChanged('email');
32          }
33      }
34      #[FakeAs(FakerMethod::DATE)]
35      public Carbon $created_at {
36          get => $this->created_at;
37          set {
38              $this->created_at = $value;
39              $this->markPropertyAsChanged('created_at');
40          }
41      }
42      #[FakeAs(FakerMethod::DATE)]
43      public Carbon $updated_at {
44          get => $this->updated_at;
45          set {
46              $this->updated_at = $value;
47              $this->markPropertyAsChanged('updated_at');
48          }
49      }
50
51      protected array $casts = [
52          'created_at' => 'datetime',
53          'updated_at' => 'datetime'
54      ];
55  }
```

Listing 10.39: model User

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 PHP8.4 mechnizmu Property Hooks pozwalającego na śledzenie prawdziwych pól bez użycia metod magicznych __get i __set

Mając tabelę i model automatycznie tworzymy fabrykę

```
php console/bin make:factory User
```

```php
<?php

declare(strict_types=1);

namespace App\Database\Factories;

use DJWeb\Framework\DBAL\Models\Factory;

class User extends Factory
{
    protected function getModelClass(): string
    {
        return \App\Database\Models\User::class;
    }

    public function definition(): array
    {
        return [
            'name' => $this->faker->name(),
            'email' => $this->faker->safeEmail(),
            'password' => $this->faker->password(),
            'created_at' => $this->faker->date(),
        ];
    }
};
```

Listing 10.40: fabryka User

Wygenerowany kod możemy dostsować

ostatnim krokiem jest stworzenie seedera i wypełnienie bazy danych

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

```php
<?php

declare(strict_types=1);

namespace App\Database\Seeders;

use App\Database\Factories\User as UserFactory;
use DJWeb\Framework\DBAL\Models\Seeder;

class UserSeeder extends Seeder
{
    public function run(): void
    {
        new UserFactory()->createMany(25);
    }
};
```

Listing 10.41: seeder UserSeed

Rozdział 26

Porównanie z istniejącymi frameworkami

Na podsumowanie książki skupimy się na porównaniu stworzonego rozwiązania z innymi popularnymi frameworkami w kilku kluczowych aspektach działania frameworka.

26.1 Minimalna wersja PHP oraz wymagane zależności

26.1.1 Stworzone rozwiązanie

Autor wyszedł z założenia, że nie będziemy się ograniczać, i ponieważ, jak zostało wielokrotnie wspomniane, korzystamy w tej książce z niemal wszystkich nowości wprowadzonych przez PHP 8.4, ta wersja języka została ustawiona jako minimalna wymagana. Poza interfejsami z PSR w `composer.json` znajduje się tylko kilka najpotrzebniejszych bibliotek:

- **Carbon** - Biblioteka do pracy z datami i czasem w PHP, oferująca intuicyjny interfejs i zaawansowane operacje, takie jak manipulacje, formatowanie i różnice między datami.

- **Symfony Mailer** - Nowoczesne narzędzie do obsługi wysyłania e-maili, wspierające różne protokoły (SMTP, Sendmail, etc.) i zapewniające bezpieczeństwo oraz łatwość konfiguracji.
- **phpdotenv** - Narzędzie do zarządzania konfiguracją aplikacji poprzez pliki `.env`, pozwalające na łatwe przechowywanie i ładowanie zmiennych środowiskowych.
- **HTMLPurifier** - Bezpieczna biblioteka do filtrowania HTML, która eliminuje potencjalnie niebezpieczne fragmenty kodu i zapewnia zgodność ze standardami.
- **Faker** - Narzędzie do generowania realistycznych, losowych danych testowych, takich jak imiona, adresy czy numery telefonów, przydatne w testach i prototypowaniu.

26.1.2 Codelgniter 4

Codelgniter 4 wymaga minimalnie PHP 7.4, co czyni go bardziej kompatybilnym z nieco starszymi środowiskami. Pod względem zależności pozostaje minimalistyczny — dostarcza gotowe narzędzia do pracy, ograniczając się do niezbędnego zestawu bibliotek wbudowanych w rdzeń frameworka.

26.1.3 Yii 2

Yii 2 jest nieco starszym frameworkiem, opartym na PHP 5.4 jako minimalnej wersji (w starszych wydaniach) oraz PHP 7.2 w najnowszych aktualizacjach. Wymaga instalacji kilku dodatkowych komponentów poprzez Composer, takich jak narzędzia ORM, autoryzacja oraz walidacja danych.

26.1.4 Laravel 11

Laravel 11 wymaga PHP 8.2 jako minimalnej wersji, co sprawia, że jest zgodny z nowoczesnymi środowiskami, ale może wymagać aktualizacji w starszych projektach. Zależności w Laravelu są rozbudowane, obejmując między innymi Eloqu-

ent ORM, Blade Template Engine, narzędzia do kolejek oraz wbudowane wsparcie dla systemów cache.

26.1.5 Symfony 7

Symfony 7 ustala PHP 8.1 jako minimalną wersję, jednocześnie dostarczając bardzo modularną strukturę. Zależności Symfony są rozbudowane, ponieważ framework ten opiera się na własnych pakietach, takich jak Symfony Components, dzięki czemu użytkownik może instalować jedynie potrzebne moduły.

26.2 Złożoność kodu

Średnia złożoność kodu we frameworku wynosi jedynie 1.3, co wskazuje na wysoką czytelność i prostotę w implementacji. Żadna z klas nie przekroczyła złożoności 10, a tylko kilka klas osiągnęło złożoność większą niż 5.

26.2.1 Codelgniter 4

Codelgniter stawia na prostotę i niską złożoność kodu. Dzięki ograniczonej liczbie funkcji i klas, jest on jednym z najłatwiejszych frameworków do nauki, co przekłada się na niską średnią złożoność.

26.2.2 Yii 2

Yii 2, ze względu na swój modułowy charakter, umożliwia osiągnięcie umiarkowanej złożoności kodu, chociaż integracja dodatkowych komponentów może podnieść ten wskaźnik.

26.2.3 Laravel 11

Laravel znany jest z bogatej funkcjonalności, co czasami skutkuje wyższą złożonością kodu. Średnia złożoność kodu w projektach opartych na Laravelu może być wyższa w porównaniu do minimalistycznych frameworków.

26.2.4 Symfony 7

Symfony jest elastyczny, ale jego złożoność kodu jest uzależniona od ilości użytych komponentów. W pełni rozwinięte aplikacje mogą mieć wyższą średnią złożoność w porównaniu do innych frameworków.

26.3 Automatyczne zależności

W rozdziale 3 została stworzona implementacja kontenera zgodna z PSR-11, zapewniająca automatyczne bindowanie parametrów konstruktorów i metod. Bazowa klasa `Application` rozszerza kontener, umożliwiając ustawianie dowolnych zależności. Ze względu na zasadę DIP z SOLID, elementy mogą być wyszukane zarówno po nazwie, jak i typie, co zwiększa elastyczność w użyciu. Dodatkowo, zastosowanie interfejsów umożliwia wydajne testowanie kolejnych modułów oraz ułatwia proces rozwijania aplikacji, zachowując przejrzystość i modularność kodu.

```
1  <?php
2
3  use DJWeb\Framework\DBAL\Connection\MySqlConnection;
4  use DJWeb\Framework\DBAL\Contracts\ConnectionContract;
5  use DJWeb\Framework\DBAL\Models\Entities\DatabaseLog;
6
7
8  $app = \DJWeb\Framework\Web\Application::getInstance();
9
10 $app->set(ConnectionContract::class, new MySqlConnection()
       );
11 /** @var MySqlConnection $connection */
```

```
$connection = $app->get(ConnectionContract::class);
$connection->connect();
$users = $connection->query('SELECT * FROM users')->
    fetchAll();
$post = $app->get(DatabaseLog::class);//automatic binding
    constructor params
```

Listing 26.1: DI - framework

26.3.1 CodeIgniter 4

CodeIgniter 4 nie posiada wbudowanego wsparcia dla pełnoprawnego kontenera zależności zgodnego z PSR-11, choć oferuje prosty mechanizm usług (`Services`), który pozwala na ręczne definiowanie i zarządzanie zależnościami w aplikacji.

```
<?php
namespace App\Controllers;

use App\Models\UserModel;

class UserController extends BaseController
{
    public function __construct()
    {
        $this->userModel = service('UserModel');
    }

    public function index()
    {
        $users = $this->userModel->findAll();
        return view('user_list', ['users' => $users]);
    }
}
```

Listing 26.2: DI - CodeIgniter

26.3.2 Yii 2

Yii 2 dostarcza mechanizm dependency injection poprzez specjalne konfiguracje w tablicach oraz umożliwia korzystanie z kontenera zależności. Jednak automatyczne bindowanie parametrów w metodach i konstruktorach wymaga dodatkowej konfiguracji.

```php
<?php
namespace app\controllers;

use Yii;
use app\models\User;

class UserController extends \yii\web\Controller
{
    private $user;

    public function __construct($id, $module, User $user,
        $config = [])
    {
        $this->user = $user;
        parent::__construct($id, $module, $config);
    }

    public function actionIndex()
    {
        $users = $this->user->find()->all();
        return $this->render('index', ['users' => $users])
            ;
    }
}
```

Listing 26.3: DI - Yii2

26.3.3 Laravel 11

Laravel 11 posiada bardzo rozbudowany kontener zależności, który pozwala na automatyczne bindowanie parametrów konstruktorów i metod. Dzięki konwencji over konfiguracji, framework automatycznie rozpoznaje zależności, a jego integracja z systemem providerów czyni go jednym z najbardziej zaawansowanych w tej dziedzinie.

```php
<?php

namespace App\Http\Controllers;

use App\Services\UserService;

class UserController extends Controller
{
    private $userService;

    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }

    public function index()
    {
        $users = $this->userService->getAllUsers();
        return view('users.index', ['users' => $users]);
    }
}
```

Listing 26.4: DI - Laravel 11

26.3.4 Symfony 7

Symfony 7 dostarcza pełnoprawny kontener zależności, zgodny z PSR-11, z wbudowanym wsparciem dla automatycznego bindowania parametrów oraz

rozpoznawania zależności na podstawie typów. Rozwiązanie to jest jednocześnie elastyczne i wydajne, pozwalając na dokładną kontrolę nad konfiguracją usług.

```php
<?php

namespace App\Controller;

use App\Service\UserService;
use Symfony\Bundle\FrameworkBundle\Controller\
   AbstractController;
use Symfony\Component\HttpFoundation\Response;

class UserController extends AbstractController
{
    private $userService;

    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }

    public function index(): Response
    {
        $users = $this->userService->getUsers();
        return $this->render('user/index.html.twig', ['
           users' => $users]);
    }
}
```

Listing 26.5: DI - Symfony 7

26.4 Routing i middleware

W rozdziale 4 stworzony został bazowy system trasowania uwzględniający najprostsze trasy. W rozdziale 12 została dodana obsługa middleware w pełni kompatybilnego z PSR-15. W rozdziale 13 router został rozbudowany o:

- Automatyczne bindowanie modeli do kontrolerów,
- Definiowanie tras przy pomocy atrybutów kontrolera,
- Grupowanie tras.

Rozdział 14 zakończył rozbudowę routera, dodając walidację formularzy opartą na atrybutach oraz automatyczne bindowanie walidatorów do kontrolerów, analogicznie jak w paczce spatie/laravel-data.

```php
<?php


use DJWeb\Framework\Auth\Auth;
use DJWeb\Framework\DBAL\Models\Entities\User;
use DJWeb\Framework\Http\Response;
use DJWeb\Framework\Routing\Attributes\RouteGroup;
use DJWeb\Framework\Routing\Attributes\RouteParam;
use DJWeb\Framework\Routing\Attributes\Route;
use DJWeb\Framework\Routing\Controller;
use DJWeb\Framework\View\Inertia\Inertia;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Server\MiddlewareInterface;
use Psr\Http\Server\RequestHandlerInterface;

//midleware definition
readonly class GuestMiddleware implements
   MiddlewareInterface
{
    public function __construct(private string $redirectTo
        = '/')
    {
    }

    public function process(ServerRequestInterface
       $request, RequestHandlerInterface $handler):
       ResponseInterface
    {
        if (Auth::check()) {
```

```
            return new Response()
                ->withHeader('Location', $this->redirectTo
                    )
                ->withStatus(303);
        }
        return $handler->handle($request);
    }
}

//config/middleware.php
return [
    'before_global' => [
        RequestLoggerMiddleware::class,
        InertiaMiddleware::class,
    ],
    'global' => [
        RouterMiddleware::class,
    ],
    'after_global' => [
        ValidationErrorMiddleware::class,
    ],
];

//sample controller

#[RouteGroup(name: 'chat')]
class ChatController extends Controller
{
    #[Route('/user/<user \d+>', methods: 'GET')]
    #[RouteParam(
        name: 'user',
        bind: User::class,
    )]
    public function user(User $user): ResponseInterface
    {
        return Inertia::render('Pages/Chat.vue', ['user'
            => $user]);
    }
}
```

Listing 26.6: routing - own framework

26.4.1 CodeIgniter 4

CodeIgniter 4 oferuje podstawowy system routingu, który pozwala na definiowanie tras w konfiguracji za pomocą tablic. Middleware w CodeIgniter 4 jest obsługiwane przez mechanizm filtrów, jednak nie jest ono zgodne z PSR-15 i może wymagać dodatkowej implementacji dla bardziej zaawansowanych zastosowań.

```php
<?php
// routes.php
$routes->get('users', 'UserController::index');
$routes->post('users/create', 'UserController::create');

// Middleware (Filters)
namespace App\Filters;

use CodeIgniter\Filters\FilterInterface;

class AuthFilter implements FilterInterface
{
    public function before(RequestInterface $request,
       $arguments = null)
    {
        if (! session()->get('isLoggedIn')) {
            return redirect()->to('/login');
        }
    }

    public function after(RequestInterface $request,
       ResponseInterface $response, $arguments = null)
    {
        // Do something after response
    }
}

// Register Filter
$routes->addFilter('auth', ['before' => ['users/*']]);
```

Listing 26.7: routing - CI4

26.4.2 Yii 2

Yii 2 posiada wbudowany system routingu, który integruje się z frameworkiem za pomocą tablic konfiguracyjnych lub reguł w plikach konfiguracyjnych. Middleware nie jest natywnie wspierane, ale można je zaimplementować przy użyciu dodatkowych komponentów lub rozszerzeń.

```php
<?php

namespace app;

// Configuring Routes
return [
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'rules' => [
                'users' => 'user/index',
                'users/create' => 'user/create',
            ],
        ],
    ],
];

// Middleware (Custom Behavior)
namespace app\components;

use yii\base\ActionFilter;

class AuthMiddleware extends ActionFilter
{
    public function beforeAction($action)
    {
        if (Yii::$app->user->isGuest) {
            return Yii::$app->response->redirect(['site/
                login']);
        }
        return parent::beforeAction($action);
    }
```

```
}

// Apply Middleware
function behaviors()
{
    return [
        'auth' => [
            'class' => AuthMiddleware::class,
        ],
    ];
}
```

Listing 26.8: routing - Yii 2

26.4.3 Laravel 11

Laravel 11 oferuje zaawansowany system routingu, trasy są domyślnie definiowane w plikach `routes/web` i `routes/api`. Middleware jest integralną częścią frameworka i pozwala na zaawansowaną obsługę żądań i odpowiedzi w aplikacji.

```
<?php
// Defining Routes
use App\Http\Controllers\UserController;

Route::get('/users', [UserController::class, 'index']);
Route::post('/users/create', [UserController::class, '
    create']);

// Middleware
namespace App\Http\Middleware;

use Closure;

class Authenticate
{
```

```
    public function handle($request, Closure $next)
    {
        if (! auth()->check()) {
            return redirect('login');
        }
        return $next($request);
    }
}


// Applying Middleware
Route::middleware(['auth'])->group(function () {
    Route::get('/users', [UserController::class, 'index']);
});
```

Listing 26.9: routing - Laravel 11

26.4.4 Symfony 7

Symfony 7 korzysta z komponentu Routing, który umożliwia definiowanie tras w plikach YAML, XML, PHP lub przy pomocy atrybutów. Middleware jest realizowane przez mechanizm EventListenerów, który jest w pełni zgodny z PSR-15, zapewniając elastyczność i wydajność.

```
<?php

// Defining Routes with Attributes
namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

class UserController
{
    #[Route('/users', name: 'user_index', methods: ['GET'
```

```
        ])]
    public function index(): Response
    {
        // Logic here
        return new Response('User list');
    }

    #[Route('/users/create', name: 'user_create', methods:
        ['POST'])]
    public function create(): Response
    {
        // Logic here
        return new Response('User created');
    }
}

// Middleware (Event Listener)
namespace App\EventListener;

use Symfony\Component\HttpKernel\Event\RequestEvent;

class AuthListener
{
    public function onKernelRequest(RequestEvent $event)
    {
        $request = $event->getRequest();
        // Authentication logic
        if (!$request->headers->has('Authorization')) {
            throw new \Symfony\Component\HttpKernel\
                Exception\UnauthorizedHttpException('Bearer
                ');
        }
    }
}

// Register Listener in services.yaml
services:
App\EventListener\AuthListener:
        tags:
            - { name: kernel.event_listener, event: kernel
                .request, method: onKernelRequest }
```

Listing 26.10: routing - Symfony 7

26.5 System zarządzania bazą danych

Ze względu na ograniczone miejsce w książce, autor zdecydował się na kompletną implementację systemu zarządzania bazą danych, lecz jedynie dla silnika MySQL. Umożliwia ona:

- System do obsługi struktury bazy danych (rozdział 7),
- Modularny *Query Builder* (rozdział 8) umożliwiający wykonywanie zaawansowanych zapytań (rozdział 8). Dzięki wzorcom projektowym, takim jak fabryka, fasada czy dekorator, system ten jest modularny i zgodny z zasadami SOLID,
- System migracji (rozdział 9) wraz z komendami pomocniczymi `make:migration` oraz `migrate`,
- ORM (rozdział 10) będący rozwiązaniem pośrednim między Doctrine i Active Record, w pełni korzystający z nowości wprowadzonych w PHP 8.4, co umożliwiło stworzenie rozwiązania zawierającego pełne typowanie encji i przypominającego Entity Framework znanego z C#. Dzięki komendzie `make:model` możliwe jest niemal automatyczne wygenerowanie modelu na podstawie bazy danych.

```
1 <?php
2 //example migration
3 namespace App\Database\Migrations;
4
5 use DJWeb\Framework\DBAL\Migrations\Migration;
6 use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
     DateTimeColumn;
7 use DJWeb\Framework\DBAL\Schema\MySQL\Columns\IntColumn;
```

```
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
   PrimaryColumn;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\TextColumn;
use DJWeb\Framework\DBAL\Schema\MySQL\Columns\
   VarcharColumn;

return new class extends Migration
{
    /**
     * run migration.
     */
    public function up(): void
    {
        $this->schema->createTable('database_logs', [
            new IntColumn('id', nullable: false,
               autoIncrement: true),
            new VarcharColumn('level'),
            new TextColumn('message'),
            new TextColumn('metadata'),
            new TextColumn('context'),
            new DateTimeColumn('created_at', current: true
               ),
            new DateTimeColumn('updated_at',
               currentOnUpdate: true),
            new PrimaryColumn('id'),
        ]);
    }

    /**
     * rollback migration.
     */
    public function down(): void
    {
        $this->schema->dropTable('database_logs');
    }
};

//example model

namespace App\Models;

```

```
namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\HasMany;
use DJWeb\Framework\DBAL\Models\Attributes\HasManyThrough;
use DJWeb\Framework\DBAL\Models\Entities\Session;
use DJWeb\Framework\DBAL\Models\Entities\User;
use DJWeb\Framework\DBAL\Models\Model;
use DJWeb\Framework\DBAL\Query\Conditions\
   WhereGroupCondition;
use DJWeb\Framework\Encryption\EncryptionService;

class Company extends Model
{
    public string $table {
        get => 'companies';
    }

    public string $name {
        get => $this->name;
        set {
            $this->name = $value;
            $this->markPropertyAsChanged('name');
        }
    }


    #[HasMany(Post::class, foreign_key: 'company_id',
       local_key: 'id')]
    public array $posts {
        get => $this->relations->getRelation('posts');
    }

    #[HasManyThrough(Comment::class, Post::class, '
       company_id', 'post_id', 'id', 'id')]
    public array $comments {
        get => $this->relations->getRelation('comments');
    }
}

//example queries
User::query()->select()
```

```
    ->where('id', '=', $userId)
    ->whereGroup(function (WhereGroupCondition $group){
        $group->where('activated', '=', 0)
            ->orWhere('mail_confirmed', '=', 0)
    })

    ->first();
Session::query()->delete()->where('id', '=', $id)->delete
    ();
new Session()->fill([
    'id' => $id,
    'payload' => new EncryptionService()->encrypt($payload
        ),
    'last_activity' => time(),
    'user_ip' => $_SERVER['REMOTE_ADDR'] ?? null,
    'user_agent' => $_SERVER['HTTP_USER_AGENT'] ?? null,
    'user_id' => null
])->save();
```

Listing 26.11: DBAL - own framework

26.5.1 CodeIgniter 4

CodeIgniter 4 oferuje wbudowany *Query Builder*, który pozwala na wykonywanie zapytań SQL w sposób zwięzły i prosty. Brakuje jednak pełnoprawnego ORM, co może wymagać od deweloperów ręcznej obsługi relacji między tabelami. System migracji jest dostępny, ale jego funkcjonalność jest bardziej ograniczona w porównaniu z nowoczesnymi frameworkami.

```
<?php

// Query Builder
$db = \Config\Database::connect();
$builder = $db->table('users');

// Insert data
```

```php
$data = [
    'name' => 'John Doe',
    'email' => 'johndoe@example.com',
];
$builder->insert($data);

// Fetch data
$query = $builder->getWhere(['id' => 1]);
$user = $query->getRow();

// Migration
namespace App\Database\Migrations;

use CodeIgniter\Database\Migration;

class AddUsersTable extends Migration
{
    public function up()
    {
        $this->forge->addField([
            'id' => [
                'type' => 'INT',
                'unsigned' => true,
                'auto_increment' => true,
            ],
            'name' => [
                'type' => 'VARCHAR',
                'constraint' => '100',
            ],
            'email' => [
                'type' => 'VARCHAR',
                'constraint' => '100',
            ],
        ]);
        $this->forge->addPrimaryKey('id');
        $this->forge->createTable('users');
    }

    public function down()
    {
        $this->forge->dropTable('users');
```

```
49      }
50  }
```

Listing 26.12: DBAL - CI4

26.5.2 Yii 2

Yii 2 wykorzystuje Active Record jako główną metodę obsługi baz danych. Jest to prostsze rozwiązanie niż Doctrine, ale mniej elastyczne w bardziej złożonych przypadkach. System migracji w Yii 2 jest dobrze zintegrowany z frameworkiem, a *Query Builder* umożliwia generowanie zaawansowanych zapytań.

```php
<?php
// Active Record
use app\models\User;

// Insert data
$user = new User();
$user->name = 'John Doe';
$user->email = 'johndoe@example.com';
$user->save();

// Fetch data
$user = User::findOne(1);

// Query Builder
$query = (new \yii\db\Query())
    ->select('*')
    ->from('users')
    ->where(['id' => 1])
    ->one();

// Migration
namespace app\migrations;

use yii\db\Migration;
```

```
25
26 class m230101_123456_add_users_table extends Migration
27 {
28     public function up()
29     {
30         $this->createTable('users', [
31             'id' => $this->primaryKey(),
32             'name' => $this->string(100)->notNull(),
33             'email' => $this->string(100)->notNull(),
34         ]);
35     }
36
37     public function down()
38     {
39         $this->dropTable('users');
40     }
41 }
```

Listing 26.13: DBAL - Yii 2

26.5.3 Laravel 11

Laravel korzysta z Eloquent ORM, który łączy prostotę Active Record z dużą elastycznością, szczególnie w obsłudze relacji. Posiada zaawansowany *Query Builder* oraz dobrze zaprojektowany system migracji, umożliwiający łatwe zarządzanie strukturą bazy danych i generowanie kodu przy użyciu komend CLI, takich jak `make:migration`.

```
1 <?php
2
3
4 // Eloquent ORM
5 use App\Models\User;
6
7 // Insert data
8 $user = new User();
```

```php
$user->name = 'John Doe';
$user->email = 'johndoe@example.com';
$user->save();

// Fetch data
$user = User::find(1);

// Query Builder
use Illuminate\Support\Facades\DB;

$user = DB::table('users')->where('id', 1)->first();

// Migration
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AddUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table
            ) {
            $table->id();
            $table->string('name', 100);
            $table->string('email', 100);
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Listing 26.14: DBAL - Laravel 11

26.5.4 Symfony 7

Symfony 7 zazwyczaj korzysta z Doctrine ORM, które oferuje ogromną elastyczność i możliwość pracy z różnymi bazami danych. Doctrine jest bardziej złożone i wymaga konfiguracji, ale oferuje potężne możliwości mapowania obiektowo-relacyjnego oraz wsparcie dla migracji i zarządzania schematem bazy danych. Symfony również wspiera *Query Builder* jako część Doctrine.

```php
<?php

// Doctrine ORM
use App\Entity\User;
use Doctrine\ORM\EntityManagerInterface;

// Insert data
$user = new User();
$user->setName('John Doe');
$user->setEmail('johndoe@example.com');
$entityManager->persist($user);
$entityManager->flush();

// Fetch data
$user = $entityManager->getRepository(User::class)->find
    (1);

// Query Builder
use Doctrine\DBAL\Query\QueryBuilder;

$queryBuilder = $connection->createQueryBuilder();
$user = $queryBuilder
    ->select('*')
    ->from('users')
    ->where('id = :id')
    ->setParameter('id', 1)
    ->execute()
    ->fetchAssociative();

// Migration
namespace DoctrineMigrations;

```

```
use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

final class Version20230101123456 extends
   AbstractMigration
{
    public function up(Schema $schema): void
    {
        $this->addSql('CREATE TABLE users (id INT
           AUTO_INCREMENT NOT NULL, name VARCHAR(100) NOT
           NULL, email VARCHAR(100) NOT NULL, PRIMARY KEY(
           id))');
    }

    public function down(Schema $schema): void
    {
        $this->addSql('DROP TABLE users');
    }
}
```

Listing 26.15: DBAL - Symfony 7

26.6 Aplikacje konsolowe

Współczesny framework to nie tylko aplikacje działające w przeglądarce, lecz również pełnoprawne aplikacje konsolowe. W rozdziale 6 została stworzona bazowa klasa komendy, wykorzystująca atrybuty do przekazywania parametrów i automatycznej rejestracji komend. Wejście i wyjście komend zostało zrealizowane przy pomocy strumieni zgodnych z PSR-7. Stworzona została bazowa komenda `MakeCommand`, która umożliwia tworzenie innych komend do automatycznego generowania elementów frameworka, takich jak modele, migracje, kontrolery itp.

```
<?php

use DJWeb\Framework\Console\Attributes\AsCommand;
use DJWeb\Framework\Console\Attributes\CommandArgument;
```

```php
use DJWeb\Framework\Console\Command;

//registered automatically
#[AsCommand('hello:world')]
class HelloWorld extends Command
{
    #[CommandArgument(name: 'world', description: 'name of
        the world to greet')]
    public protected(set) string $world = 'Earth';

    public function run(): int
    {
        $this->getOutput()->info("Hello, {$this->world}!")
            ;
        return 0;
    }
}
```

Listing 26.16: Console - own framework

26.6.1 CodeIgniter 4

CodeIgniter 4 posiada system konsolowy o ograniczonej funkcjonalności w porównaniu z nowoczesnymi frameworkami. Komendy można definiować ręcznie, ale nie są one automatycznie rejestrowane ani nie korzystają z bardziej zaawansowanych rozwiązań, takich jak atrybuty czy PSR-7.

```php
<?php
namespace App\Commands;

use CodeIgniter\CLI\BaseCommand;
use CodeIgniter\CLI\CLI;

class HelloWorld extends BaseCommand
{
    protected $group       = 'demo';
```

```
    protected $name        = 'hello:world';
    protected $description = 'Outputs Hello World!';

    public function run(array $params)
    {
        $name = $params[0] ?? 'World';
        CLI::write("Hello, $name!");
    }
}

// Register the command in Config/Commands.php
public $commands = [
    'hello:world' => 'App\Commands\HelloWorld',
];
```

Listing 26.17: Console - CI4

26.6.2 Yii 2

Yii 2 wspiera aplikacje konsolowe poprzez komponent `yii\console\Controller`. Komendy są definiowane jako klasy kontrolerów konsolowych, a parametry mogą być przekazywane jako argumenty CLI. Brakuje jednak automatycznej rejestracji oraz bardziej nowoczesnego podejścia, jak wprowadzone przez PSR.

```
<?php
namespace app\commands;

use yii\console\Controller;

class HelloController extends Controller
{
    public $name = 'World';

    public function options($actionID)
    {
        return ['name'];
```

```
    }

    public function actionIndex()
    {
        echo "Hello, {$this->name}!\n";
    }
}
```

Listing 26.18: Console - Yii 2

26.6.3 Laravel 11

Laravel 11 posiada zaawansowany system aplikacji konsolowych oparty na pakiecie Artisan. Komendy są definiowane za pomocą klas, które dziedziczą z `Illuminate\Console\Command`. Parametry można łatwo definiować w klasach, a system automatycznej rejestracji oraz wsparcie dla generowania kodu przy użyciu komend takich jak `make:model` czy `make:controller` czyni Artisan jednym z najbardziej rozwiniętych rozwiązań.

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class HelloWorld extends Command
{
    protected $signature = 'hello:world {name=World}';
    protected $description = 'Outputs Hello World!';

    public function handle()
    {
        $name = $this->argument('name');
        $this->info("Hello, $name!");
```

```
    }
}

// Register the command in app/Console/Kernel.php
protected $commands = [
    \App\Console\Commands\HelloWorld::class,
];
```

Listing 26.19: Console - Laravel 11

26.6.4 Symfony 7

Symfony 7 wykorzystuje komponent `Console`, który jest jednym z najbardziej uniwersalnych rozwiązań do budowy aplikacji konsolowych. Komendy są definiowane jako klasy z pełnym wsparciem dla zależności z kontenera DI oraz automatyczną rejestracją parametrów. Komponent ten jest również w pełni kompatybilny z PSR-7.

```
<?php

namespace App\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

class HelloWorldCommand extends Command
{
    protected static $defaultName = 'app:hello-world';

    protected function configure()
    {
        $this
            ->setDescription('Outputs Hello World!')
            ->addArgument('name', InputArgument::OPTIONAL,
```

```
                'Name of the person', 'World');
19      }
20
21      protected function execute(InputInterface $input,
            OutputInterface $output): int
22      {
23          $name = $input->getArgument('name');
24          $output->writeln("Hello, $name!");
25          return Command::SUCCESS;
26      }
27  }
```

Listing 26.20: Console - Symfony 7

26.7 Widoki

Stworzony w książce system renderowania widoków jest bardzo uniwersalny i umożliwia podpięcie dowolnego istniejącego mechanizmu do renderowania widoków. Korzystając z tego interfejsu, w rozdziale 15 podłączono Twig, w rozdziale 16 stworzony został własny adapter na pliki Blade, natomiast w rozdziale 17 zdecydowano się na nowoczesne aplikacje webowe i własny adapter Inertia.js oparty na plikach Blade w połączeniu z biblioteką Vue 3.

```
1   <?php
2
3   use DJWeb\Framework\Routing\Attributes\Route;
4   use DJWeb\Framework\Routing\Attributes\RouteGroup;
5   use DJWeb\Framework\Routing\Controller;
6   use DJWeb\Framework\View\Inertia\Inertia;
7
8   #[RouteGroup('views')]
9   class ControllerRenderingBlade extends Controller
10  {
11      #[Route('blade')]
12      public function blade(): ResponseInterface
13      {
```

```
14          $this->withRenderer('blade');
15          return $this->render('index.blade.php', ['user' =>
                'test', 'x' => 3]);
16      }
17
18      #[Route('twig')]
19      public function twig(): ResponseInterface
20      {
21          $this->withRenderer('twig');
22          return $this->render('index.twig', ['user' => '
                test']);
23      }
24
25      #[Route('/inertia', 'GET')]
26      public function index(): ResponseInterface
27      {
28          return Inertia::render('InertiaRendering.vue', [
29              'test' => 'test'
30          ]);
31      }
32
33  }
```

Listing 26.21: View - own framework

26.7.1 CodeIgniter 4

CodeIgniter 4 oferuje prosty system widoków oparty na standardowych plikach PHP, bez wsparcia dla zaawansowanych szablonów takich jak Blade czy Twig. Brak wbudowanej obsługi nowoczesnych narzędzi, takich jak Inertia.js, czyni go mniej elastycznym w stosunku do współczesnych wymagań.

```
1   <?php
2   // Controller
3   namespace App\Controllers;
4
```

```
class Home extends BaseController
{
    public function index()
    {
        return view('welcome_message', ['title' => '
            Welcome to CodeIgniter!']);
    }
}

// View (app/Views/welcome_message.php)
<!DOCTYPE html>
<html>
<head>
    <title><?= esc($title) ?></title>
</head>
<body>
<h1><?= esc($title) ?></h1>
<p>Hello, CodeIgniter!</p>
</body>
</html>
```

Listing 26.22: View - CI4

26.7.2 Yii 2

Yii 2 wykorzystuje system widoków oparty na plikach PHP, z opcjonalnym wsparciem dla szablonów Twig. Framework umożliwia także rozszerzenie mechanizmu widoków, co pozwala na integrację z innymi systemami szablonów, jednak wymaga to dodatkowej konfiguracji.

```
<?php
// Controller
namespace app\controllers;

use Yii;
use yii\web\Controller;
```

```
7
8  class SiteController extends Controller
9  {
10     public function actionIndex()
11     {
12         return $this->render('index', ['title' => 'Welcome
               to Yii!']);
13     }
14 }
15
16 // View (views/site/index.php)
17 <!DOCTYPE html>
18 <html>
19 <head>
20     <title><?= htmlspecialchars($title, ENT_QUOTES) ?></
          title>
21 </head>
22 <body>
23 <h1><?= htmlspecialchars($title, ENT_QUOTES) ?></h1>
24 <p>Hello, Yii 2!</p>
25 </body>
26 </html>
```

Listing 26.23: View - Yii 2

26.7.3 Laravel 11

Laravel 11 korzysta z szablonów Blade, które są szybkie, lekkie i bardzo elastyczne. Framework oferuje również wsparcie dla Inertia.js, co czyni go idealnym wyborem dla nowoczesnych aplikacji webowych, łącząc backend PHP z frontendem opartym na Vue 3, React lub Svelte.

```
1  <?php
2
3
4  // Controller
```

```
namespace App\Http\Controllers;

class HomeController extends Controller
{
    public function index()
    {
        return view('welcome', ['title' => 'Welcome to
            Laravel!']);
    }
}

// View (resources/views/welcome.blade.php)
<!DOCTYPE html>
<html>
<head>
    <title>{{ $title }}</title>
</head>
<body>
    <h1>{{ $title }}</h1>
    <p>Hello, Laravel Blade!</p>
</body>
</html>

// Example with Inertia.js
return Inertia::render('Home', ['title' => 'Welcome to
   Laravel with Inertia!']);
\end{verbatim}
```

Listing 26.24: View - Laravel 11

26.7.4 Symfony 7

Symfony 7 wykorzystuje domyślnie Twig jako mechanizm szablonów, oferując zaawansowane możliwości renderowania widoków. Dzięki swojej modularnej strukturze Symfony umożliwia integrację z innymi mechanizmami, ale nie posiada natywnego wsparcia dla systemów takich jak Inertia.js, które muszą być ręcznie zaimplementowane.

```php
<?php
// Controller
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\
   AbstractController;
use Symfony\Component\HttpFoundation\Response;

class HomeController extends AbstractController
{
    public function index(): Response
    {
        return $this->render('home/index.html.twig', [
            'title' => 'Welcome to Symfony!',
        ]);
    }
}

// Twig Template (templates/home/index.html.twig)
<!DOCTYPE html>
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ title }}</h1>
    <p>Hello, Symfony Twig!</p>
</body>
</html>
```

Listing 26.25: View - Symfony 7

26.8 Obsługa wyjątków

Obsługa wyjątków stworzona w książce (rozdział 18) składa się z trzech podstawowych składników:

- Wykorzystania bazowych funkcji PHP do rejestracji handlerów błędów i wyjątków,
- Interfejsu przeglądarkowego opartego na Vue 3, co pozwala na czytelne zobaczenie w trybie developerskim, co się stało,
- Interfejsu konsolowego będącego pełnoprawną aplikacją konsolową, umożliwiającą debugowanie wyjątków w konsoli, między innymi poprzez przeglądanie `backtrace`.

26.8.1 Codelgniter 4

Codelgniter 4 posiada prosty mechanizm obsługi błędów, który generuje raporty w formacie HTML dla trybu developerskiego oraz logi dla trybu produkcyjnego. Brakuje bardziej zaawansowanych narzędzi, takich jak konsolowy debuger czy interfejsy wizualne oparte na frameworkach frontendowych.

26.8.2 Yii 2

Yii 2 oferuje wbudowany system obsługi wyjątków z możliwością wyświetlania szczegółowych raportów błędów w trybie developerskim, w tym widoków przedstawiających szczegóły `backtrace`. Brakuje jednak natywnego wsparcia dla interfejsów frontendowych opartych na technologiach takich jak Vue 3.

26.8.3 Laravel 11

Laravel 11 korzysta z `Ignition`, zaawansowanego narzędzia do obsługi wyjątków, które oferuje czytelne raporty błędów z interfejsem przeglądarkowym. Nie posiada jednak wbudowanego konsolowego debugera błędów, co sprawia, że debugging w CLI wymaga dodatkowych narzędzi.

26.8.4 Symfony 7

Symfony 7 zawiera zaawansowany system obsługi błędów, który generuje szczegółowe raporty w trybie developerskim, w tym wizualne podsumowanie wyjątków w przeglądarce. Integracja z komponentem `Debug` pozwala na dokładną analizę `backtrace`, ale konsolowy debuger nie jest częścią standardowego zestawu narzędzi.

26.9 Sesja i ciastka

Stworzone zostały handlery do sesji oparte na bazie danych i plikach, pokazując zalety i wady obu rozwiązań. Oba rozwiązania korzystają z nowoczesnego szyfrowania danych opartego na dostępnym od PHP 7.2 algorytmie `sodium`. Dane przechowywane w cookies również są szyfrowane. Dodatkowo, dzięki komendzie `key:generate`, możliwe jest stworzenie klucza szyfrującego i zapisanie go w pliku `.env`.

26.9.1 CodeIgniter 4

CodeIgniter 4 obsługuje sesje za pomocą plików, bazy danych lub pamięci cache, z opcją szyfrowania danych sesji. Obsługa ciastek jest prostsza, bez natywnego wsparcia dla zaawansowanego szyfrowania, co wymaga dodatkowej konfiguracji lub bibliotek.

26.9.2 Yii 2

Yii 2 oferuje obsługę sesji i ciastek z możliwością przechowywania danych w plikach, bazie danych lub pamięci cache. Dane w ciasteczkach mogą być szyfrowane przy użyciu wbudowanych narzędzi, jednak mechanizm ten wymaga ręcznego skonfigurowania kluczy szyfrowania.

26.9.3 Laravel 11

Laravel 11 wspiera sesje przechowywane w plikach, bazie danych, pamięci cache, Redisie i innych systemach. Ciasteczka są domyślnie szyfrowane za pomocą klucza generowanego komendą `key:generate`, co zapewnia wysokie bezpieczeństwo. Framework oferuje również prosty sposób na zarządzanie kluczami szyfrującymi w pliku `.env`.

26.9.4 Symfony 7

Symfony 7 obsługuje sesje przechowywane w plikach, bazie danych lub pamięci cache, z opcją szyfrowania. Obsługa ciasteczek w Symfony wymaga ręcznego skonfigurowania szyfrowania za pomocą komponentu `Security`, co daje dużą elastyczność, ale może być bardziej skomplikowane w konfiguracji.

26.10 Wysyłka maili

Stworzony w książce system opakowuje, zgodnie z zasadami SOLID, niskopoziomową bibliotekę `symfony/mailer`, umożliwiając także wysyłkę maili HTML opartą na plikach Blade lub Twig. Rozwiązanie to zapewnia elastyczność i modularność, pozwalając na łatwe dostosowanie do potrzeb aplikacji.

```
1  <?php
2
3  //php console/bin WelcomeMailable
4
5  class WelcomeMailable extends Mailable
6  {
7      public function __construct(
8          private User $user
9      ) {}
10
11     public function content(): Content
12     {
13         return new Content('mail/welcome.blade.php', [
```

```
            'username' => $this->user->username
        ]);
    }

    public function envelope(): Envelope
    {
        return new Envelope(
            from: new Address(Config::get('mail.from.address'), Config::get('mail.from.name')),
            subject: 'Welcome to ' . Config::get('app.name'),
        )->addTo( new Address($this->user->email, $this->user->username));
    }
}

//sending
MailerFactory::createSmtpMailer(...Config::get('mail.default'))
    ->send(new WelcomeMailable($event->user));
```

Listing 26.26: Mail - own framework

26.10.1 CodeIgniter 4

CodeIgniter 4 posiada wbudowaną klasę `Email`, która obsługuje wysyłkę wiadomości e-mail za pomocą protokołów SMTP, Mail i Sendmail. Brakuje jednak wsparcia dla nowoczesnych szablonów HTML, co wymaga dodatkowych integracji.

```
<?php

// Configuring Email
$email = \Config\Services::email();

$email->setFrom('noreply@example.com', 'CodeIgniter App');
```

```
$email->setTo('user@example.com');
$email->setSubject('Welcome to CodeIgniter!');
$email->setMessage('<p>Hello, this is a plain HTML email
    .</p>');

// Sending Email
if ($email->send()) {
    echo 'Email sent successfully!';
} else {
    echo $email->printDebugger();
}
```

Listing 26.27: Mail - CI4

26.10.2 Yii 2

Yii 2 wspiera wysyłkę maili za pomocą komponentu `yii mailer`, który pozwala na korzystanie z różnych adapterów, takich jak SwiftMailer. Obsługa maili HTML jest możliwa, ale integracja z szablonami Twig czy Blade wymaga dodatkowej konfiguracji.

```
<?php


// Using Yii Mailer
Yii::$app->mailer->compose()
    ->setFrom('noreply@example.com')
    ->setTo('user@example.com')
    ->setSubject('Welcome to Yii!')
    ->setHtmlBody('<p>Hello, this is a plain HTML email.</
       p>')
    ->send();

// Optionally using Twig templates
return [
    'components' => [
```

```
        'mailer' => [
            'viewPath' => '@app/mail',
            'useFileTransport' => false,
            'view' => [
                'class' => 'yii\twig\ViewRenderer',
            ],
        ],
    ],
];
```

Listing 26.28: Mail - Yii2

26.10.3 Laravel 11

Laravel 11 oferuje zaawansowany system wysyłki maili oparty na klasie `Illuminate\Mail`, który wspiera różne systemy dostarczania wiadomości, takie jak SMTP, Mailgun czy AWS SES. Laravel natywnie wspiera szablony Blade do generowania treści HTML wiadomości.

```
<?php

use Illuminate\Support\Facades\Mail;

// Sending a simple email
Mail::to('user@example.com')->send(new \App\Mail\
    WelcomeMail());

// Mail class with Blade template (App/Mail/WelcomeMail.
    php)
namespace App\Mail;

use Illuminate\Mail\Mailable;

class WelcomeMail extends Mailable
{
```

```
    public function build()
    {
        return $this->from('noreply@example.com', 'Laravel
            App')
            ->subject('Welcome to Laravel!')
            ->view('emails.welcome', ['name' => 'User']);
    }
}

// Blade template (resources/views/emails/welcome.blade.
   php)
<!DOCTYPE html>
<html>
<body>
    <h1>Welcome to Laravel!</h1>
    <p>Hello, {{ $name }}. This is a welcome email.</p>
</body>
</html>
```

Listing 26.29: Mail - Laravel 11

26.10.4 Symfony 7

Symfony 7 wykorzystuje komponent `Mailer`, który zapewnia wysoką elastyczność i wsparcie dla różnych protokołów wysyłki e-mail. Dzięki integracji z komponentem `Twig`, możliwe jest tworzenie zaawansowanych szablonów HTML, jednak brakuje natywnej obsługi innych silników szablonów, takich jak Blade.

```
<?php

use Symfony\Component\Mailer\Mailer;
use Symfony\Component\Mailer\Transport;
use Symfony\Component\Mime\Email;

// Configure mailer
```

```
$transport = Transport::fromDsn('smtp://localhost');
$mailer = new Mailer($transport);

// Create and send email
$email = (new Email())
    ->from('noreply@example.com')
    ->to('user@example.com')
    ->subject('Welcome to Symfony!')
    ->html('<p>Hello, this is a plain HTML email.</p>');

$mailer->send($email);

// Using Twig for email templates
use Symfony\Bridge\Twig\Mime\TemplatedEmail;

$email = (new TemplatedEmail())
    ->from('noreply@example.com')
    ->to('user@example.com')
    ->subject('Welcome to Symfony!')
    ->htmlTemplate('emails/welcome.html.twig')
    ->context(['name' => 'User']);

// Twig template (templates/emails/welcome.html.twig)
<!DOCTYPE html>
<html>
<body>
    <h1>Welcome to Symfony!</h1>
    <p>Hello, {{ name }}. This is a welcome email.</p>
</body>
</html>
```

Listing 26.30: Mail - Symfony 7

26.11 Autoryzacja

Stworzony system autoryzacji jest dość prosty – opiera się na klasie `User` oraz prostym systemie ról i uprawnień. Przykładowe widoki do rejestracji czy przy-

pominania hasła zostały wykonane w Vue 3, zapewniając nowoczesny interfejs użytkownika.

26.11.1 CodeIgniter 4

CodeIgniter 4 nie posiada wbudowanego systemu autoryzacji, ale oferuje narzędzia, takie jak `Filters`, które można wykorzystać do implementacji systemu ról i uprawnień. Wiele funkcjonalności wymaga jednak użycia dodatkowych bibliotek lub ręcznego wdrożenia.

26.11.2 Yii 2

Yii 2 oferuje system `RBAC` (Role-Based Access Control), który umożliwia zarządzanie rolami i uprawnieniami. System jest elastyczny, ale jego konfiguracja może być bardziej złożona w porównaniu do prostszych frameworków.

26.11.3 Laravel 11

Laravel 11 posiada wbudowany system autoryzacji oparty na politykach (`Policies`) i bramkach (`Gates`), co umożliwia szczegółowe kontrolowanie dostępu do zasobów. Obsługuje także system ról i uprawnień przy użyciu dodatkowych pakietów oraz oferuje gotowe szablony widoków dla rejestracji i zarządzania użytkownikami.

26.11.4 Symfony 7

Symfony 7 korzysta z komponentu `Security`, który pozwala na zaawansowaną kontrolę dostępu opartą na rolach oraz specjalnych regułach. System ten jest bardzo elastyczny, ale może wymagać znacznego nakładu konfiguracji, szczególnie w prostszych przypadkach użycia.

26.12 Cache

System do obsługi danych podręcznych jest w pełni zgodny z PSR-6. Stworzono storage oparte na zwykłych plikach oraz z wykorzystaniem Redisa. Dodatkowo zaimplementowane zostały strategie zarządzania pamięcią oraz funkcje takie jak `Cache::remember`, umożliwiające zapisanie w pamięci podręcznej dowolnego callbacka, co znacząco upraszcza pracę z danymi podręcznymi.

26.12.1 CodeIgniter 4

CodeIgniter 4 oferuje prosty system cache, obsługujący pliki, Redis, Memcached oraz inne popularne systemy przechowywania danych podręcznych. Brakuje jednak zgodności z PSR-6, co ogranicza możliwości przenoszenia kodu między projektami.

26.12.2 Yii 2

Yii 2 posiada wbudowany system cache z obsługą różnych mechanizmów, takich jak pliki, Redis czy Memcached. Chociaż system jest wydajny i elastyczny, nie jest zgodny z PSR-6, co może utrudniać integrację z zewnętrznymi bibliotekami.

26.12.3 Laravel 11

Laravel 11 oferuje zaawansowany system cache, wspierający pliki, Redis, Memcached i inne mechanizmy. Funkcje takie jak `Cache::remember` są natywnie obsługiwane, co czyni ten system bardzo przyjaznym dla deweloperów. Laravel nie posiada natywnej zgodności z PSR-6, ale może być rozszerzony, aby ją osiągnąć.

26.12.4 Symfony 7

Symfony 7 korzysta z komponentu Cache, który jest w pełni zgodny z PSR-6 i PSR-16. Oferuje wsparcie dla różnych systemów przechowywania danych podręcznych oraz zaawansowane mechanizmy zarządzania pamięcią, co czyni go jednym z najbardziej elastycznych rozwiązań.

26.13 System eventów

Stworzony w książce system obsługi eventów stanowi prostą implementację PSR-14, zapewniającą zgodność z nowoczesnymi standardami PHP. W centrum systemu znajduje się klasa EventManager, która pełni rolę fasady, upraszczając zarządzanie eventami. Eventy wraz z ich listenerami są definiowane w pliku konfiguracyjnym, co ułatwia konfigurację i rozwój aplikacji.

```php
<?php

// Event class
readonly class UserRegisteredEvent
{
    public function __construct(
        public User $user
    ) {}
}

// Listener class
class SendWelcomeEmailListener
{
    public function __invoke(UserRegisteredEvent $event):
       void
    {
        MailerFactory::createSmtpMailer(...Config::get('
           mail.default'))
            ->send(new WelcomeMailable($event->user));
    }
}

```

```php
// EventManager configuration (config/events.php)
return [
    UserRegisteredEvent::class => [
        SendWelcomeEmailListener::class,
    ],
];

// Dispatching the event
new \DJWeb\Framework\Events\EventManager()->dispatch(new
   UserRegisteredEvent($user));
```

Listing 26.31: Events - own framework

26.13.1 CodeIgniter 4

CodeIgniter 4 posiada prosty system eventów, który pozwala na rejestrowanie i wywoływanie listenerów. System jest intuicyjny, ale brak zgodności z PSR-14 może stanowić ograniczenie w bardziej zaawansowanych projektach.

```php
<?php
// Registering an event and listener
Events::on('userRegistered', function ($user) {
    echo "User registered: {$user->email}";
});

// Triggering the event
Events::trigger('userRegistered', $user);
```

Listing 26.32: Events - CI4

26.13.2 Yii 2

Yii 2 oferuje wbudowany system eventów, który umożliwia rejestrowanie handlerów oraz integrację z cyklem życia aplikacji. System nie jest zgodny z PSR-14 i wymaga rejestracji eventów bezpośrednio w kodzie, co może utrudniać skalowalność.

```php
<?php

// Registering an event handler
Event::on(
    User::class,
    User::EVENT_AFTER_INSERT,
    function ($event) {
        echo "User registered: {$event->sender->email}";
    }
);

// Triggering the event
$user = new User();
$user->save(); // Automatically triggers
   EVENT_AFTER_INSERT
```

Listing 26.33: Events - Yii2

26.13.3 Laravel 11

Laravel 11 korzysta z systemu eventów, który pozwala na definiowanie listenerów i eventów w dedykowanych klasach. Chociaż system ten nie implementuje PSR-14, jest bardzo elastyczny i dobrze zintegrowany z pozostałymi częściami frameworka, np. kolejkami (queues).

```php
<?php

class UserRegistered
```

```
{
    public function __construct(public $user) {}
}

// Listener class
class SendWelcomeEmail
{
    public function handle(UserRegistered $event)
    {
        Mail::to($event->user->email)->send(new
            WelcomeMail());
    }
}

// Registering in EventServiceProvider
protected $listen = [
    UserRegistered::class => [
        SendWelcomeEmail::class,
    ],
];

// Dispatching the event
event(new UserRegistered($user));
```

Listing 26.34: Events - Laravel 11

26.13.4 Symfony 7

Symfony 7 wykorzystuje komponent `EventDispatcher`, który jest zgodny z PSR-14 i oferuje zaawansowaną obsługę eventów. Listenerów można rejestrować w plikach konfiguracyjnych, co czyni ten system bardzo elastycznym i zgodnym z nowoczesnymi standardami.

```
<?php

// Event class
```

```php
class UserRegisteredEvent
{
    public function __construct(public $user) {}
}

// Listener class
class SendWelcomeEmailListener
{
    public function __invoke(UserRegisteredEvent $event)
    {
        Mailer::send('welcome_email', ['user' => $event->user]);
    }
}

// services.yaml
services:
    App\EventListener\SendWelcomeEmailListener:
        tags:
            - { name: 'kernel.event_listener', event: 'user.registered' }

// Dispatching the event
$dispatcher->dispatch(new UserRegisteredEvent($user), 'user.registered');
```

Listing 26.35: Events - Symfony 7

26.14 Kolejki

Mechanizm stworzony w książce implementuje zarówno obsługę jobów, jak i schedulera. Joby dziedziczą po abstrakcyjnej klasie `Job`, która dzięki wykorzystaniu atrybutu `#[SerializeTo]` pozwala na pełną kontrolę automatycznej serializacji. Mechanizm obsługuje backendy oparte na Redisie oraz bazie danych, co zapewnia elastyczność w zależności od wymagań projektu.

```php
<?php

//example job
namespace Tests\Helpers;

use DJWeb\Framework\Config\Config;
use DJWeb\Framework\Log\Log;
use DJWeb\Framework\Scheduler\Attributes\Serialize;
use DJWeb\Framework\Scheduler\Job;

class TestJob extends Job
{
    public function __construct(
        #[Serialize]
        public protected(set) string $id,
        #[Serialize('custom_title')]
        public protected(set) string $title,
        #[Serialize]
        public protected(set) string $name = 'John Doe',
        #[Serialize('custom_email')]
        public protected(set) string $email = '
            john@example.com',
        #[Serialize]
        public protected(set) int $age = 30
    ) {
        parent::__construct();
    }

    public function handle(): void
    {
        Log::info('job processed');
    }

    public function handleException(\Throwable $e): void
    {
        Log::error('job failed', context: ['trace' => $e->
            getTraceAsString()]);
    }
}

//sending
```

```
40 QueueFactory::make('redis')->push(new TestJob(id: '1',
       title: 'job'));
```

Listing 26.36: Jobs - own framework

26.14.1 CodeIgniter 4

CodeIgniter 4 nie posiada wbudowanego mechanizmu kolejek, co wymaga użycia zewnętrznych bibliotek lub ręcznego wdrożenia obsługi kolejek. Brak wsparcia dla popularnych backendów, takich jak Redis czy bazy danych, jest ograniczeniem w bardziej zaawansowanych projektach.

```
1 <?php
2 // CodeIgniter does not have built-in queue support.
3 // Developers need to implement their own solutions or use
      external libraries.
```

Listing 26.37: Jobs - CI4

26.14.2 Yii 2

Yii 2 oferuje system kolejek jako rozszerzenie, które wspiera różne backendy, takie jak Redis, RabbitMQ czy baza danych. Joby mogą być definiowane jako klasy, jednak brak natywnej obsługi serializacji wymaga dodatkowej konfiguracji lub użycia zewnętrznych bibliotek.

```
1 <?php
2
3 // Define a job
4 class SendEmailJob extends \yii\base\BaseObject implements
      \yii\queue\JobInterface
```

```
{
    public $email;
    public $subject;
    public $body;

    public function execute($queue)
    {
        Yii::$app->mailer->compose()
            ->setTo($this->email)
            ->setSubject($this->subject)
            ->setTextBody($this->body)
            ->send();
    }
}

// Push job to queue
Yii::$app->queue->push(new SendEmailJob([
    'email' => 'user@example.com',
    'subject' => 'Welcome!',
    'body' => 'Thank you for joining us!',
]));
```

Listing 26.38: Jobs - Yii2

26.14.3 Laravel 11

Laravel 11 posiada zaawansowany system kolejek, który wspiera wiele backendów, takich jak Redis, AWS SQS czy RabbitMQ, a także bazy danych. Joby są definiowane jako klasy dziedziczące po `Illuminate\Queue\Job`, a framework natywnie obsługuje ich serializację oraz integrację z systemem schedulera.

```
<?php


```

```php
// Define a job
class SendEmailJob implements \Illuminate\Contracts\Queue\
    ShouldQueue
{
    use \Illuminate\Bus\Queueable;

    public function __construct(
        public string $email,
        public string $subject,
        public string $body
    ) {}

    public function handle(): void
    {
        Mail::to($this->email)->send(new WelcomeMail($this
            ->subject, $this->body));
    }
}

// Dispatching a job
SendEmailJob::dispatch('user@example.com', 'Welcome!', '
    Thank you for joining us!');

// Scheduling a job
$schedule->job(new SendEmailJob('user@example.com', 'Daily
    Update', '...'))
    ->dailyAt('08:00');
```

Listing 26.39: Jobs - Laravel 11

26.14.4 Symfony 7

Symfony 7 korzysta z komponentu `Messenger`, który oferuje obsługę kolejek oraz przesyłania wiadomości. Mechanizm wspiera różne backendy, w tym Redis i bazy danych, oraz umożliwia pełną kontrolę nad serializacją i deserializacją jobów. Scheduler wymaga jednak osobnej konfiguracji lub dodatkowych narzędzi.

```php
<?php

use Symfony\Component\Messenger\MessageBusInterface;

// Define a job
class SendEmailJob
{
    public function __construct(
        private string $email,
        private string $subject,
        private string $body
    ) {}

    public function handle(): void
    {
        // Logic for sending email
    }
}
```

Listing 26.40: Jobs - Symfony 7

26.15 WebSocket

Zaimplementowany w ramach frameworka własny serwer WebSocket jest oparty na niskopoziomowej bibliotece `react/event-loop`. Serwer umożliwia obsługę dwukierunkowej komunikacji w czasie rzeczywistym między klientem a serwerem. Jego uruchomienie jest możliwe za pomocą komendy **ws:run**. Rozwiązanie to doskonale integruje się z opisywanym wcześniej frontendem wykonanym przy pomocy Vue 3, umożliwiając tworzenie nowoczesnych aplikacji w czasie rzeczywistym.

26.15.1 CodeIgniter 4

CodeIgniter 4 nie posiada wbudowanego wsparcia dla WebSocket. Wdrożenie tej technologii wymaga ręcznej integracji z zewnętrznymi bibliotekami, takimi jak Ratchet czy ReactPHP, co może być pracochłonne.

26.15.2 Yii 2

Yii 2 również nie dostarcza natywnego wsparcia dla WebSocket. Możliwe jest jednak wykorzystanie dodatkowych rozszerzeń, takich jak `yiisocket`, które pozwalają na implementację serwerów WebSocket, choć wymaga to większego nakładu pracy.

26.15.3 Laravel 11

Laravel 11 korzysta z technologii Reverb, która umożliwia komunikację w czasie rzeczywistym między serwerem a klientem. Jednak Reverb ogranicza się do komunikacji typu server-to-client, co oznacza, że nie obsługuje pełnoprawnej dwukierunkowej komunikacji WebSocket. Dla bardziej zaawansowanych zastosowań WebSocket wymaga użycia dodatkowych pakietów, takich jak `laravel-websockets`.

26.15.4 Symfony 7

Symfony 7 nie posiada natywnego wsparcia dla WebSocket, ale można wykorzystać komponenty, takie jak `Ratchet` lub integracje z ReactPHP, aby wdrożyć serwery WebSocket. Takie podejście wymaga jednak ręcznej konfiguracji i dodatkowych zasobów.

Spis listingów

Skorowidz