DAMIAN JÓŹWIAK

SOLID FRAMEWORK PHP

PHP 8.4 IN PRACTICE: BUILDING NEXT-GENERATION APPLICATIONS AND FRAMEWORKS

PHP 8.4 in Practice Building Next-Generation Applications and Frameworks

Author: **Damian Jóźwiak**

Cover design: **Maciej Pieda - Pieda Art Design**

First Edition: **2024**

Book's website: **`https://en.masterphp.eu`**

Source code available:
`https://github.com/DJWeb-Damian-Jozwiak/book`

Disclaimer

The framework created in this book is fully covered by automated tests, ensuring that each of its functions has been thoroughly verified for accuracy. The author, based on his current knowledge, considers the framework to be stable and safe. However, it should be noted that this publication was created for educational purposes and serves more as a proof of concept for creating a custom framework, with a detailed explanation of how its core internal components work.

Despite the high quality and safety of the code, the author wishes to clearly emphasize that he **does not recommend** using this framework in real production environments. The framework was developed primarily as an educational tool, and its use in production environments involves risks that each user assumes at their own responsibility.

Therefore, the author **does not accept responsibility** for any potential issues arising from the use of this framework in production environments, such as functional errors, system failures, or possible data loss.

Welcome to a book that merges technology with passion, and code with storytelling. On the following pages, you will find both practical guidelines and personal reflections that accompanied me throughout the process of creating this framework. To demonstrate that coding is not just the art of algorithms, but also a journey that each of us undertakes in our own way, this book includes two unconventional introductions.

The first, inspired by the Spanish song *„La historia de un niño"* by Porta, pays tribute to the universal nature of programming and the personal path of development. The second, in a lighthearted and humorous tone, refers to the iconic opening of the series *„Walker, Texas Ranger"* and is dedicated to all those who face daily programming challenges with determination and a smile.

This book is not just a guide to the framework, but also a **„modern-day PHP bible"** – a compendium of contemporary standards, best practices, and inspiration for building software. Within its pages, you will find everything that makes PHP one of the most powerful and versatile programming languages. From PSR compliance to a modular approach to code, each chapter demonstrates how to build tools fit for the 21st century.

Both of these inspirations are united by the belief that behind every line of code lies a story, and programming is not just a tool, but also a way to express oneself. Let's begin this journey!

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

Contents

Chapter 1

Introduction

In this chapter, we will introduce key concepts, briefly discuss best practices in software engineering, and present design patterns that have been applied. We will also explain what PSR is, why it plays a significant role in modern applications, and discuss selected standards. At the end of the chapter, we will configure the working environment, preparing the repository and testing tools such as phpunit and phpstan.

1.1 Best Practices

Many programming best practices have been devised in software engineering. When creating your own framework, the SOLID and DRY principles will be particularly useful.

The DRY principle ("Don't Repeat Yourself") is crucial in object-oriented programming because it promotes writing code that is more readable, easier to maintain, and less prone to errors. Avoiding code duplication means that each piece of information or functionality is defined only once, which facilitates making changes and updates without the risk of inconsistency. In the context of object-oriented programming, DRY encourages the use of mechanisms such as inheritance, composition, and polymorphism to efficiently manage shared behaviors and properties among objects, leading to more flexible and scalable code.

1.1.1 SOLID

SOLID is an acronym for five object-oriented design principles invented by Robert C. Martin. The letters correspond to the following principles:

- **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)

Let us consider the following classes:

```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;
}
```

```
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: SOLID Principles

Single-Responsibility Principle - SRP

A class should have only one responsibility. The only thing the `Square` and `Circle` classes do is calculate their area. The responsibility of the `AreaCalculator` class is to calculate the area for an array of shapes, regardless of how each shape calculates its area. Meanwhile, the only responsibility of the `SumCalculatorOutputter` class is to output the result as text or JSON.

Open-Closed Principle - OCP

Classes should be open for extension but closed for modification. Since the `Square` and `Circle` classes use an interface to calculate their area, adding a new class (e.g., a triangle) does not require modifying the calculator class. Additionally, using `array_map` ensures that all objects will have a getter implemented for the `$area` property.

Liskov Substitution Principle - LSP

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing the specific details of those objects. The `BigAreaCalculator` class adheres to this principle because it does not significantly matter what the base class does as long as it maintains compatibility with the inherited class.

Interface Segregation Principle - ISP

Many specialized interfaces are better than one general-purpose interface. The `SumCalculatorOutputter` class follows this principle because each type of output is handled by a dedicated interface.

Dependency Inversion Principle - DIP

High-level modules should not depend on low-level modules; the dependencies between them should arise from abstractions. Each of the modules adheres to this principle. The calculators take a shape interface in their constructors, while the output display class takes a calculator interface in its constructor. It's also important to note that all classes receive their dependencies through the constructor. We will leverage this in Chapter Three, where we will implement a Dependency Injection container that forms the foundation of our framework.

1.1.2 YAGNI

YAGNI stands for `You Aren't Gonna Need It`. In practice, this means that we only add necessary dependencies to the `composer.json` file. In our case, these will include:

- PSR interfaces,
- the `Carbon` library for date management,

- the `Faker` library for generating sample data and model factories,
- the `Symfony/Mailer` library for low-level email handling.

This approach helps minimize the risk of potential security vulnerabilities and avoids unnecessary code.

1.2 Design Patterns

Design patterns are proven solutions to common problems in object-oriented programming. They help organize code in a way that is readable, flexible, and easy to maintain. In this section, we will look at a few popular design patterns that play a key role in creating scalable and efficient software.

1.2.1 Builder

The Builder pattern is used to construct complex objects step by step. It is particularly useful when an object has many optional parameters or requires complicated initialization. By using the Builder, you can separate the construction process of an object from its representation, making it easier to modify and add new options.

```
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 Decorator

The Decorator pattern allows for dynamically extending the functionality of objects without needing to modify their code. The Decorator is often used to add new behaviors to a class in a flexible way, without relying on inheritance. This avoids the proliferation of subclasses and simplifies code maintenance. This pattern is extremely useful in achieving the first SOLID principle and keeping class cyclomatic complexity at a level of 5 or fewer.

```
<?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: Decorator Pattern

1.2.3 Factory

The Factory pattern allows for the creation of objects without directly using the `new` operator. Instead, the entire process is managed by a factory method, providing flexible control over the created instances and avoiding tight coupling between classes.

```
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: Factory Pattern

1.2.4 Facade

The Facade pattern provides a simplified interface to a more complex system of classes or a complicated API. The Facade hides the complexity of the system and allows clients easier access to its functionality without requiring a deeper understanding of internal details. This pattern will be used frequently in the framework we are developing. The main class of a given module will usually serve as a facade for many other classes that make up the module.

```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: Facade Pattern

1.2.5 Composite

The Composite pattern allows for representing a hierarchy of objects in the form of a tree. It enables treating individual objects and groups of objects in the same way. This is useful when we want to build complex objects composed of many simpler components.

```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: Composite Pattern

1.2.6 Singleton

The Singleton pattern ensures that a class has only one instance and provides global access to it. This is particularly useful when global access to a resource is required, such as a database connection or logging. Singleton ensures control over object creation and prevents multiple instances. The `Web/Application` and `Console/Application` classes that we will create in Chapters Four and Six will be excellent examples of this pattern, as we will only need one instance of these classes across the entire framework, depending on whether we are working in a web or console environment.

```
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) is a set of recommendations aimed at standardizing coding style and practices within the PHP ecosystem. Each standard defined within PSR impacts different aspects of application development, ranging from code organization to more complex topics such as HTTP request management and the use of design patterns. PSR is essential because it ensures consistency and predictability across various projects, making it easier for developers to collaborate and integrate external libraries.

Each PSR standard is developed by the PHP-FIG (PHP Framework Interoperability Group) and is widely adopted throughout the PHP ecosystem. Understanding these standards and implementing them is crucial for building modern, scalable applications.

All of the standards listed below will be utilized in the framework we are creating, except for PSR-12, which will be adhered to in all code listings.

1.3.1 PSR-3, Creating Logs

PSR-3 defines an interface for logging in PHP applications. This standard introduces a set of methods that can be implemented by various logging libraries,

allowing for interchangeable use of different implementations depending on the project's needs. The main methods of PSR-3 include `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info`, and `debug`.

Implementing PSR-3 allows for better monitoring of an application's performance, identifying potential issues, and analyzing its behavior in production environments. Due to the consistent interface, the logging mechanism can be easily swapped with another one without requiring code modifications, ensuring greater flexibility and modularity.

1.3.2 PSR-4 and Autoloading

PSR-4 introduces a standard for class autoloading in PHP, which is crucial for efficient code organization in larger projects. Thanks to PSR-4, each class in a project is mapped to a directory structure, allowing for automatic class loading without the need for manual `require` or `include` statements.

Using PSR-4 greatly simplifies code management and supports better organization of namespaces (`namespace`). This also enables easier integration with external libraries and frameworks since all namespaces are clearly defined and can be loaded automatically.

In the context of large projects, such as MVC (Model-View-Controller) frameworks, following PSR standards, particularly PSR-4, is very important. MVC frameworks are complex and consist of many components. Standards like PSR-4 help maintain organized code, making it easier to navigate and understand the project structure.

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, HTTP Requests

PSR-7 defines a standard for managing HTTP requests and responses in PHP applications. This standard introduces interfaces for classes that represent both HTTP requests and responses, allowing easy manipulation during application processing. The key concepts associated with PSR-7 are `RequestInterface`, `ResponseInterface`, `StreamInterface`, and `UriInterface`.

PSR-7 is often used in web applications, especially those based on the middleware pattern, where each request is processed at various levels. Thanks to standardization, applications can easily integrate different libraries that also implement this standard.

1.3.5 PSR-11, Adding a Dependency Container

PSR-11 defines an interface for Dependency Injection Containers. A dependency container is a key component in the architecture of many PHP applications, enabling efficient management of dependencies between classes. Thanks to PSR-11, various dependency containers can be interchangeably used in applications, provided they implement the required interfaces.

Using PSR-11 makes the code more modular and testable. The ease of managing dependencies allows for dynamically supplying different class implementations, which increases project flexibility and enhances the separation of concerns.

1.3.6 PSR-12, Code Quality

PSR-12 extends and refines previous recommendations regarding coding style in PHP, building upon the standards introduced by PSR-1 and PSR-2. It provides detailed guidelines on code formatting, such as the use of spaces, the structure of class and function declarations, and the formatting of control blocks.

The introduction of PSR-12 aims to improve code readability and make it easier for different developers to review the code. Adherence to this standard ensures that the code is more consistent, which, in the long term, improves its maintainability.

1.3.7 PSR-14, Event Dispatcher

PSR-14 defines interfaces for an event mechanism, enabling PHP applications to use the Observer design pattern. This standard specifies how events can be registered, dispatched, and processed within an application. The event mechanism facilitates better internal communication within the application, especially in complex projects where various modules must respond to specific events.

Using PSR-14 allows for loose coupling between application components, making it easier to introduce new features and modifications without disrupting the structure of existing code.

1.3.8 PSR-15, Adding Middleware

PSR-15 introduces a standard for middleware in PHP applications. Middleware are layers that mediate the processing of HTTP requests, enabling tasks such as authorization, data validation, or request modification. PSR-15 defines interfaces for middleware classes, allowing for the creation of consistent and easily extensible solutions.

By implementing PSR-15, it is possible to create modular applications in which individual layers of logic can be added or modified without needing to interfere with the entire application code. Middleware is a crucial component in modern PHP frameworks like Symfony and Laravel.

1.3.9 PSR-17, HTTP Factory

PSR-17 defines interfaces for factories that create HTTP-related objects, such as requests (`RequestFactoryInterface`), responses (`ResponseFactoryInterface`), and streams (`StreamFactoryInterface`). This standard allows for the creation of HTTP objects in a consistent and flexible manner, which is especially useful in large applications that must dynamically handle various types of requests.

PSR-17 unifies the process of creating HTTP objects, enhancing interoperability between different libraries and frameworks. This approach promotes building flexible and scalable web solutions.

1.4 Dependency Management

Composer is a dependency management tool in PHP that allows you to declare the libraries your project depends on and manage their installation and updates.

1.4.1 Dependency Management

Composer is not a package manager in the traditional sense, like Yum or Apt. Instead, it manages libraries at the project level, installing them in a directory (e.g., `vendor`) within the project. By default, nothing is installed globally, making Composer a dependency manager. However, for convenience, global installation is possible using the `global` command.

This idea is not new, and Composer draws strong inspiration from tools like npm for Node.js or Bundler for Ruby.

1.4.2 Usage Example

Suppose you have a project that depends on several libraries. Some of these libraries also have their own dependencies.

Composer allows you to:

- declare the libraries your project depends on,
- specify which versions of packages can and should be installed, and then install them (i.e., downloading them to the project),
- update all dependencies with a single command.

More information can be found at: `https://getcomposer.org/doc/00-intro.md`

1.5 Local Repository Setup

1.5.1 Creating the Directory Structure

First, let's create the basic directory structure for the project. In the console (or terminal), execute the following commands:

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

1.5.2 Creating the Main Project File

Navigate to the `public` directory and create the `index.php` file:

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

You can edit this file and add some basic PHP code, for example:

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

1.5.3 Creating an Empty Repository in the Framework Directory

In the framework directory, create an empty Git repository:

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

1.5.4 Adding the composer.json File

In the framework directory, create the composer.json file:

```
[language=bash] touch composer.json
```

In the composer.json file, define the minimal structure:

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

1.5.5 Adding the symfony/var-dumper Package

The symfony/var-dumper package offers more advanced and readable variable output compared to the standard var_dump and print_r functions in PHP. var_dump and print_r display data structures without formatting, which can be difficult to read when dealing with complex structures like multi-dimensional arrays or objects. In contrast, symfony/var-dumper formats

data more clearly, adding colors and structural indentation, making it easier to analyze and debug the code.

Navigate to the main project directory and create the `composer.json` file:

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

To add the framework repository and the `symfony/var-dumper` package, include the following content in the `composer.json` file:

```
[language=JSON]
{
    "repositories": [
        {
            "type": "path",
            "url": "./framework"
        }
    ],
    "require": {
        "my_project/framework": "*",
        "symfony/var-dumper": "^5.0"
    }
}
```

1.5.6 Dependency Installation

To install the dependencies, run the following command in the terminal:

```
 composer install
```

This command will download all required packages and install them in the `vendor` directory.

1.5.7 vHost Configuration

Configuring virtual hosts (vHosts) is essential for hosting multiple websites on a single server. Below are example configuration files for Apache and Nginx. Ensure that the paths specified in the files correspond to where the directories were created in the steps above. Before starting the configuration, make sure that PHP 8.4 FPM is installed:

```
sudo apt update
sudo apt install php8.4-fpm
```

Add to the `/etc/hosts` file (or `C:\Windows\System32\drivers\etc\hosts file in Windows`)

```
[language=bash]
127.0.0.1 framework.local
```

Apache

Created config file for Apache:

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

Add following config:

```
<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>
```

Activate config and reload Apache

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

Nginx Create config for vhost

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

Add following config:

```
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;
}
```

Activate vhost and reload nginx

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

```
sudo systemctl reload nginx
```

1.6 Application Entry Point

The configuration presented in this chapter assumes a single application entry point— the `public/index.php` file.

Modern web applications often use a single entry point, typically the `public/index.php` file. This approach, popular in many frameworks such as Laravel, Symfony, and Zend, offers several advantages that contribute to its widespread use.

1.6.1 Advantages of a Single Entry Point

The first and most important advantage of a single entry point is the centralization of web traffic management. All incoming traffic is directed to one file, making it easy to manage routing, authorization, and other aspects of the application. This enables efficient control over access to different resources and ensures consistency in request handling.

The second key advantage is security. Having a single entry point allows for the implementation of global security mechanisms, such as input filtering and protection against SQL Injection or XSS attacks. Additionally, it simplifies the deployment of server-level security policies, such as restricting access to specific files or directories.

1.7 PHPUnit

PHPUnit is one of the most popular unit testing frameworks for PHP. It allows writing and running unit tests, which help ensure that individual parts of the application work as expected.

The basic principles of testing:

- **Arrange:** Setting up the test environment, including creating objects and initializing data.
- **Act:** Performing the operation to be tested.
- **Assert:** Verifying whether the result of the operation matches the expected outcome.

1.8 phpstan

phpstan is a static analysis tool for PHP code that helps detect errors without needing to run the code. phpstan analyzes code at various levels of detail, from 0 to 9, where each successive level is more restrictive and thorough.

phpstan levels:

- **Level 0:** Basic analysis, detecting the simplest errors.
- **Levels 1-4:** Gradually increasing strictness, checking variable types, method compatibility, etc.
- **Levels 5-8:** More detailed analysis, verifying more advanced use cases.
- **Level 9:** The most restrictive analysis, full type checking, and code compliance verification.

1.9 PHPUnit and phpstan configuration

In order to add PHPUnit and PHPStan to project follow those steps Step 1: PHPUnit installation

```
composer require --dev phpunit/phpunit:^11
```

Step 2:PHPStan installation

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

Krok 3: Autoloading config In ‘composer.json‘ file you need to add ‘autoload‘ i ‘autoload-dev‘: sections

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

Next you need to update autoloaded classes:

```
composer dump-autoload
```

Krok 4: PHPStan - configuration Create ‘phpstan.neon‘ file in project root directory:

```
parameters:
    level: 8
    paths:
        - src
        - tests
```

Krok 5: PHPUnit - configurationt Create ‘phpunit.xml‘file in project root directory:

```
<phpunit bootstrap="vendor/autoload.php">
```

```
    <testsuites>
        <testsuite name="Unit Tests">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

1.9.1 Xdebug and Code Coverage

Xdebug is a PHP extension that enables debugging and profiling of code. In the context of unit testing, Xdebug is often used to generate code coverage reports.

Installing Xdebug To install Xdebug, follow these steps:

```
pecl install xdebug
```

Next, add Xdebug to the php.ini file:

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

Generating a Code Coverage Report To generate a code coverage report, run PHPUnit with the appropriate options:

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

The code coverage report will be saved in the coverage directory.

Chapter 10

DBAL - ORM

In the world of web application programming, models play a key role in representing data and its structure. Models are an abstraction that allows us to work with data in a more organized and understandable way. They help us better reflect real-world business objects and complex relationships between them without directly manipulating the database. In other words, models serve as a "bridge" between the object-oriented world in which we operate and the relational databases used by our applications.

Models are indispensable because they simplify the application-building process, making it easier to manage data and their relationships. Thanks to them, a developer can focus on business logic rather than technical details related to database manipulation.

10.1 ORM – Object-Relational Mapping

ORM (Object-Relational Mapping) is a design pattern that enables the mapping of data from relational databases to objects in application code. The main goal of ORM is to simplify database interaction, eliminating the need for writing complex SQL queries. ORM allows mapping tables and columns to classes and object properties, making data management in an application more intuitive and aligned with the object-oriented programming paradigm. In the PHP world, two popular ORM approaches are Doctrine and Active Record.

10.1.1 Doctrine vs. Active Record – A Brief Comparison

Doctrine is a more complex and flexible ORM tool that introduces the Unit of Work pattern and strong separation of the data layer from business logic. With Doctrine, we have full control over the mapping process and can precisely define data types, relationships between entities, and operations performed on data. Doctrine enables very complex compound operations but requires more knowledge and configuration from the developer.

Active Record, on the other hand, is simpler and more straightforward to use. This pattern integrates data access logic directly with the model. As a result, the developer can perform CRUD (Create, Read, Update, Delete) operations directly on model instances, speeding up and simplifying application development. The advantages of Active Record are its intuitiveness and ease of use, particularly in smaller projects where the flexibility of Doctrine is not required. The downside is lower scalability and limited control over more advanced data operations.

10.1.2 PHP 8.4: Revolution with Property Hooks

PHP 8.4 introduces the Property Hooks mechanism, comparable to a similar solution long known from the C# language. In C#, this mechanism allows easy tracking of changes in object properties and executing additional logic upon their modification. The introduction of Property Hooks to PHP opens up new possibilities in the context of ORM, especially for the simplified Active Record approach.

With Property Hooks, it becomes possible to automatically track changes in model fields, enabling the implementation of the Active Record pattern with a NotifyPropertyChangesContract interface. This contract would allow the ORM to be automatically informed of any changes without the need for manual update or data validation handling. This significantly simplifies data management in models.

Moreover, attributes introduced in PHP 8.0 allow defining additional metadata, such as data types or relationships between entities, similar to Doctrine. This allows combining the intuitiveness of Active Record with the more ad-

vanced capabilities of defining data structures known from Doctrine. These new PHP mechanisms make designing models not only easier but also more flexible, suitable for both smaller and more complex applications.

10.2 Field Observer

The first step is creating a class that observes changes in fields (PropertyObserver class). Its main task is to maintain a reference to an abstract Model class, encompassing all created models. This class primarily serves to collect information about modified fields in the form of an array, where the keys are field names, and the values are their current values. If this array contains the primary key, it indicates the necessity of performing an update in the database. Otherwise, it is assumed that the model has not yet been saved in the database and needs to be inserted.

```php
<?php

namespace DJWeb\Framework\DBAL\Models;

use DJWeb\Framework\DBAL\Models\Contracts\
    NotifyPropertyChangesContract;

class PropertyObserver implements
    NotifyPropertyChangesContract
{
    public function __construct(private Model $model)
    {
    }
    /**
     * @var array<string, int|string|float|null>
     */
    private array $changedProperties = [];

    public function markPropertyAsChanged(
        string $propertyName,
        float|int|string|null $value,
    ): 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: PropertyObserver class

10.3 Abstract Model Class

The next step is creating an abstract Model class that will meet several key requirements and functions. First, each class inheriting from this abstraction will need to override the getter for the `table` field, allowing the determination of which database table the model should be assigned to. We will also introduce a function that leverages the new PHP 8.4 feature of overriding getters for individual class fields. This function, called `markPropertyAsChanged`, will be responsible for registering changes in model fields by recording this information in the observer. Every model will use this method in its setters to mark which fields require saving to the database.

In our database, each field is stored as a string. However, using Typed Properties available since PHP 7.4, we can better manage data types in our framework. To enable type conversion, we will add a `Castable` interface containing a single `from(string)` method that returns an instance of itself. This will allow seamless mapping of database values to enum types and any other

objects implementing this interface. Additionally, we will provide out-of-the-box support for casting dates to Carbon objects, significantly simplifying working with date data in the system.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Contracts;

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

Listing 10.2: Castable interface

```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; }
```

```
18
19      protected(set) string $primary_key_name = 'id';
20      private(set) PropertyObserver $observer;
21
22      public int|string $id {
23          get => $this->id;
24          set {
25              $this->id = $value;
26              $this->markPropertyAsChanged('id');
27          }
28      }
29      /**
30       * @var array<string, string>
31       */
32      protected array $casts = [];
33
34      public function __construct()
35      {
36          $this->observer = new PropertyObserver($this);
37      }
38
39      public function markPropertyAsChanged(string
            $property_name): void
40      {
41          $this->observer->markPropertyAsChanged(
42              $property_name,
43              $this->$property_name
44          );
45      }
46
47      public bool $is_new {
48          get => $this->observer->is_new;
49      }
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: Model class

10.4 Extending QueryBuilder

To build a CRUD operation mechanism, we will use a DI container that returns the appropriate query builder (select, create, update, delete). The `ModelQueryBuilder` class we create will serve as a facade for these operations. We will add a public readonly field for the facade and a private `builder` field, which will be one of the four query builders.

Key implementation elements:

- The `select` method will be overridden to default the model's table to the builder.
- The `get` and `first` methods will retrieve data using the base builder and then map it to the model using `hydrate` and `hydrateMany` functions.
- The `hydrate/hydrateMany` functions will populate the model using the `fill` method, which utilizes previously created casting mechanisms.

Below is the `fill` method of the Model class:

```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;
    }
}
```

Listing 10.4: fill function in Model class

The implementation of the discussed QueryBuilder for models is as follows

```php
<?php

```

```php
declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\QueryBuilders;

use DJWeb\Framework\DBAL\Contracts\Query\
    QueryBuilderFacadeContract;
use DJWeb\Framework\DBAL\Models\Model;
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
    ;

class ModelQueryBuilder
{
    public readonly QueryBuilderFacadeContract $facade;
    private SelectQueryBuilder|UpdateQueryBuilder|
        InsertQueryBuilder|DeleteQueryBuilder $builder;

    public function __construct(protected(set) Model
        $model)
    {
        $this->facade = new QueryBuilder();
    }

    public function select($columns = ['*']): static
    {
        $this->builder = $this->facade->select($this->
            model->table);
        $this->builder->select($columns);
        return $this;
    }

    public function first(): ?Model
    {
        $result = $this->builder->first();
        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: ModelQueryBuilder class

10.5 Adding Data to the Database

To enable saving data to the database in accordance with SOLID principles and using the combined approaches of Doctrine and Active Record patterns, we will create a mechanism consisting of three classes: EntityInserter, EntityUpdater, and EntityManager. This division of responsibility will allow for effective management of saves, both through the model class and the dedicated manager.

The `EntityManager` will act as a facade that decides whether to invoke an insert or update operation on the model, depending on whether the model already exists in the database.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Decorators;

use DJWeb\Framework\DBAL\Models\Model;

class EntityManager
{
    private readonly EntityUpdater $updater;
    private readonly EntityInserter $inserter;
    public function __construct(private Model $model)
    {
        $this->updater = new EntityUpdater($this->model);
        $this->inserter = new EntityInserter($this->model)
            ;
    }

    public function save(): void
    {
        if ($this->model->observer->is_new) {
            $this->model->id = $this->inserter->insert();
        } else {
            $this->updater->update();
        }
    }
```

```
}
```

Listing 10.6: EntityManager class

EntityInserter and EntityUpdater will handle the appropriate operations for inserting and updating data in the database.

Implementation of the EntityInserter class:

```
<?php

declare(strict_types=1);

namespace DJWeb\Framework\DBAL\Models\Decorators;

use DJWeb\Framework\DBAL\Contracts\Query\
   QueryBuilderFacadeContract;
use DJWeb\Framework\DBAL\Models\Model;
use DJWeb\Framework\DBAL\Models\PropertyObserver;

class EntityInserter
{
    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 insert(): ?string
    {
        $builder = $this->query_builder->insert($this->
           model->table);
        $builder->values($this->property_watcher->
```

```
            getChangedProperties())
29           ->execute();
30
31       return $builder->getInsertId();
32   }
33 }
```

Listing 10.7: EntityInserter class

Implementation of the EntityUpdater class:

```
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 EntityUpdater
12 {
13     private QueryBuilderFacadeContract $query_builder;
14     private PropertyObserver $property_watcher;
15     public function __construct(
16
17         private Model $model
18     ) {
19         $this->query_builder = $this->model->query_builder
               ->facade;
20         $this->property_watcher = $this->model->observer;
21     }
22
23     public function update(): void
24     {
25         $this->query_builder->update($this->model->table)
26             ->set($this->property_watcher->
```

```
                getChangedProperties());
27      }
28  }
```

Listing 10.8: EntityUpdater class

The model will be decorated with a decorator, which in its `save` method will call `save` from the EntityManager. This way, saving will be possible directly from the model level.

```
1   <?php
2
3   abstract class Model implements PropertyChangesContract
4   {
5       private EntityManager $entity_manager;
6
7       public function __construct()
8       {
9           $this->entity_manager = new EntityManager($this);
10      }
11
12      public function save(): void
13      {
14          $this->entity_manager->save();
15      }
16  }
```

Listing 10.9: Saving model to the database

10.6 Adding Relationships

To implement relationship handling in our framework, we will use four basic relationships.

- BelongsTo
- HasMany
- HasManyThrough
- BelongsToMany

All will be implemented using modern PHP features, such as attributes available since PHP 8. This will ensure consistency in the framework, which already uses these mechanisms to define commands, parameters, and field getters (since PHP 8.4).

10.7 HasMany Attribute

HasMany: This relationship indicates that one record in a table (e.g., a company) can have many related records in another table (e.g., posts). In the example, the HasMany relationship between Company and Post is defined using an attribute, and the getter returns the result of the `getRelation('posts')` method, which retrieves all posts related to the company, using the company_id and id keys.

```
1  <?php
2
3  namespace Tests\Helpers\Models;
4
5  use DJWeb\Framework\DBAL\Models\Attributes\HasMany;
6  use DJWeb\Framework\DBAL\Models\Model;
7
8  class Company extends Model
9  {
10     public string $table {
11         get => 'companies';
12     }
13
14     #[HasMany(Post::class, foreign_key: 'company_id',
       local_key: 'id')]
```

```
    public array $posts {
        get => $this->relations->getRelation('posts');
    }
}
```

Listing 10.10: HasMany relationship example

10.8 BelongsTo Attribute

BelongsTo: This relationship works in the opposite direction – it indicates that a given record belongs to a single related record in another table. In the Post class, BelongsTo is a relationship with Company. The getter for the `company` field returns the appropriate instance of the Company class by fetching data from the database based on the foreign key `company_id`.

```
<?php

namespace Tests\Helpers\Models;

use DJWeb\Framework\DBAL\Models\Attributes\BelongsTo;
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');
        }
    }

```

```
22      #[BelongsTo(Company::class, foreign_key: 'company_id',
            local_key: 'id')]
23      public Company $company {
24          get {
25              /** @var Company $model */
26              $model = $this->relations->getRelation('
                    company');
27              return $model;
28          }
29      }
30  }
```

Listing 10.11: BelongsTo relationship example

The use of field getters (available since PHP 8.4) enables lazy loading, which means that the related data will only be retrieved when accessing the field that represents the relationship. For example, retrieving all posts for a specific company happens only when you refer to the `$posts` field in the Company class. This relationship dynamically generates the query: `SELECT * FROM posts WHERE company_id = ?`

10.8.1 HasManyThrough Attribute

HasManyThrough: a relationship through an intermediate model. We use it when we want to access models that are related through an intermediate model.

An example of an SQL query illustrating this relationship:

```
1  CREATE TABLE projects (
2                              id INT PRIMARY KEY,
3                              name VARCHAR(255)
4  );
5
6  CREATE TABLE departments (
7                                  id INT PRIMARY KEY,
8                                  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: has many through relationship in sql

10.8.2 BelongsToMany Attribute

BelongsToMany - many-to-many relationship. We use it when there is a many-to-many relationship between two models through a pivot table.

An example of an SQL query illustrating this relationship:

```
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: belongs to many relationship in sql

10.8.3 Abstract Relation Class

To build a relationship mechanism based on an abstract `Relation` class, we can approach it in a way that maintains the consistency and flexibility of the entire framework. The `Relation` class will be responsible for fetching data from the database in the form of an array. Derived classes such as `HasMany` and `BelongsTo` will need to enforce specific behaviors, such as generating appropriate SQL queries, setting `where` conditions, and returning models.

The abstract `Relation` class will handle general operations related to executing queries and fetching data as arrays from the database. Derived classes, like `HasMany` and `BelongsTo`, will need to implement specific methods such as setting appropriate `where` conditions and transforming data into models.

```
<?php

declare(strict_types=1);

```

```php
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,
        protected string $foreign_key,
        protected string $local_key,
    ) {
        $this->query = $this->createQueryBuilder();
    }
    abstract protected function createQueryBuilder():
       QueryBuilderContract;
    abstract public function addConstraints(): void;

    /**
     * @return array<int, Model>|Model
     */
    public function getResults(): array|Model {
        return $this->query->select()->get();
    }

    abstract public function getRelated(string $property):
        array|Model;
}
```

Listing 10.14: Relation class

10.8.4 HasMany Relationship

In a `HasMany` relationship, the SQL query will include a `where` condition that searches for all records in the related table where the foreign key matches the value of the local key.

```php
<?php

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 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
```

```
32      {
33          $results = $this->getResults();
34          return array_map(
35              fn (array $result) => new $this->related()->
                    fill($result),
36              $results
37          );
38      }
39  }
```

Listing 10.15: HasMany class

10.8.5 BelongsTo Relationship

In a BelongsTo relationship, the SQL query searches for a record in the parent table where the local key matches the foreign key value in the child model.

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\DBAL\Models\Relations;
6
7   use DJWeb\Framework\DBAL\Contracts\Query\
        QueryBuilderContract;
8   use DJWeb\Framework\DBAL\Models\Model;
9   use DJWeb\Framework\DBAL\Models\Relation;
10
11  class BelongsTo extends Relation
12  {
13      protected function createQueryBuilder():
            QueryBuilderContract
14      {
15          return $this->parent->query_builder->facade->
                select(
16              $this->related::getTable()
17          );
```

```
18      }
19
20      public function addConstraints(): void
21      {
22          $this->query->where(
23              $this->local_key,
24              '=',
25              $this->parent->{$this->foreign_key}
26          );
27      }
28
29      /**
30       * @param string $property
31       * @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: BelongsTo class

10.8.6 HasManyThrough Relationship

The HasManyThrough relationship inherits from the HasMany relationship, only overriding the relationship conditions using the JOIN operator.

```
1   <?php
2
3   declare(strict_types=1);
4
5   namespace DJWeb\Framework\DBAL\Models\Relations;
6
7   use DJWeb\Framework\DBAL\Models\Model;
```

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

class HasManyThrough extends HasMany
{
    /**
     * @param Model $parent
     * @param class-string<Model> $related
     * @param class-string<Model> $through
     * @param string $first_key
     * @param string $second_key
     * @param string $local_key
     * @param string $second_local_key
     */
    public function __construct(
        protected Model  $parent,
        protected string $related,
        protected string $through,
        protected string $first_key,
        protected string $second_key,
        protected string $local_key,
        protected string $second_local_key
    )
    {
        parent::__construct($parent, $related, $first_key,
            $local_key);
    }

    public function addConstraints(): void
    {
        $this->query
            ->innerJoin(
                table: $this->through::getTable(),
                first: $this->through::getTable() . '.' .
                   $this->second_local_key,
                operator: '=',
                second: $this->related::getTable() . '.' .
                    $this->second_key
            )
            ->where(
                $this->through::getTable() . '.' . $this->
                   first_key,
```

```
                    '=',
                    $this->parent->id
                );
        }
}
```

Listing 10.17: HasManyThrough class

10.8.7 BelongsToMany Relationship

The `BelongsToMany` relationship inherits from the `HasMany` relationship, only overriding the relationship conditions using the `JOIN` operator.

```
<?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,
```

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

Listing 10.18: BelongsToMany class

10.8.8 Relationship Factory

To simplify the creation of relationships based on model attributes, we can use the Factory pattern. The relationship factory will have two static methods that will take the model and the attribute as parameters and dynamically create the appropriate relationships (HasMany, BelongsTo, etc.) based on this. This approach will automate the process of creating relationships, making the framework more flexible. The factory will use PHP 8 attributes to read relationship

information stored in the model.

```php
<?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,
```

```
34          BelongsToManyAtrribute $attribute,
35      ): RelationContract {
36          $relation = new BelongsToMany(
37              $parent,
38              $attribute->related,
39              $attribute->pivot_table,
40              $attribute->foreign_pivot_key,
41              $attribute->related_pivot_key
42          );
43          $relation->addConstraints();
44          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: RelationFactory class

The relationship factory provides a consistent and flexible way to create relationships, regardless of their type. This allows the framework to easily handle HasMany, BelongsTo, and other relationship types. With attributes and the factory, the process of creating relationships is automated. The user only needs to declare the attributes in the models, and the rest is handled by the framework.

10.8.9 Relationship Decorator

The relationship handling mechanism in our framework relies on a decorator for the Model class, which manages relationships using two private arrays serving as simplified caches. One array stores information about declared relationships, while the other stores values already retrieved from the database. This mechanism allows relationships to be created and managed dynamically, without the need to repeatedly process the same data. When the `getRelation` method is called in the derived class, the framework parses the attributes, then uses the relationship factory to generate the appropriate query, fetches the data, and stores it in the cache, significantly improving performance.

A key role in this mechanism is played by the asynchronous visibility of fields, available since PHP 8.4. Thanks to this, the `relations` field has visibility set to `protected private(set)`, meaning it is marked as readonly in derived classes, ensuring safe access to relationships, but in the base class, it can be set at any time, for example, when loading relationships from the database. This approach eliminates the need to create additional getters or method mappings, keeping the code simple and readable while maintaining full control over the relationship mechanism in models.

```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;
```

```php
use ReflectionProperty;

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

    /**
     * @var array<string, Model|array<int|string, mixed>>
     */
    private array $relationsCache = [];

    public function __construct(private readonly Model
        $model)
    {
    }

    /**
     * @param string $name
     *
     * @return Model|array<int|string, Model>|null
     *
     * @throws \ReflectionException
     */
    public function getRelation(string $name): Model|array
        |null
    {
        if (! isset($this->relations[$name])) {
            $property = new ReflectionProperty($this->
                model, $name);
            $this->initializeRelation($property);
        }
        $exception = new \RuntimeException("Relation {
            $name} not found");
        $this->relationsCache[$name]
            ??= $this->relations[$name]?->getRelated() ??
                throw $exception;
        return $this->relationsCache[$name];
    }
```

```
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(
```

```
                $property,
                $attribute->newInstance()
            )
        );
    }

    private function initializeBelongsToMany(
        ReflectionProperty $property,
        BelongsToManyAttribute $attribute
    ): void {
        $value = RelationFactory::belongsToMany($this->model, $attribute);
        $this->relations[$property->getName()] = $value;
    }

    private function initializeBelongsTo(
        ReflectionProperty $property,
        BelongsToAttribute $attribute
    ): void {
        $value = RelationFactory::belongsTo($this->model, $attribute);
        $this->relations[$property->getName()] = $value;
    }

    private function initializeHasManyThrough(
        ReflectionProperty $property,
        HasManyThroughAttribute $attribute
    ): void {
        $value = RelationFactory::hasManyThrough($this->model, $attribute);
        $this->relations[$property->getName()] = $value;
    }

    private function initializeHasMany(
        ReflectionProperty $property,
        HasManyAttribute $attribute
    ): void {
        $value = RelationFactory::hasMany($this->model, $attribute);
        $this->relations[$property->getName()] = $value;
    }
```

```
125 }
```

Listing 10.20: RelationFactory class

10.9 Adding a Factory

In test environments, it is crucial to quickly populate the database with appropriate test data. Instead of creating a data generation mechanism from scratch, we will use the mature and popular Faker/Faker library, which offers methods to generate various types of data in many languages. This allows us to easily generate fake data such as names, addresses, phone numbers, and much more, significantly simplifying the test preparation process. To add it to the project, run `composer require fakerphp/faker`.

As part of our solution, we will create a base model factory that will accept an array of attributes and automatically populate the model using its `fill` method. This will allow for quick creation of model instances with test data. To further simplify this process, we will add the `MakeFactory` command, which will make it easy to generate factories for different models, enabling the rapid and efficient preparation of test data throughout the project.

10.9.1 How It Works

The abstract `Factory` class will be the foundation for model factories, and its implementation is simple and efficient. In the constructor, this class accepts a `Generator` instance from the Faker library, which allows generating test data. The class will provide several methods: `make`, `create`, and `createMany`, which automate the process of creating and filling models with test data.

Main methods of the `Factory` class:

- `make`: Creates an instance of the model and fills it with generated data, but does not save it to the database. This allows working with the model without immediately saving it to the database.

- `create`: Creates the model, fills it with data, and then saves it to the database.
- `createMany`: Creates multiple instances of models in a loop, using the `range` language construct to determine the number of instances, and then saves each model to the database.

```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;
    }

```

```
33      /**
34       * @param int $count
35       * @param array<int, Model> $attributes
36       * @return array
37       */
38      public function createMany(int $count, array
            $attributes = []): array
39      {
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: Factory class

10.9.2 MakeFactory Command

Although modern PHP offers advanced data typing mechanisms, creating a class that automatically generates a factory based on a model does not need to be complicated. We can achieve this using template (stub) mechanisms, enums, and attributes, which greatly simplify the process of defining model attributes and linking them with the corresponding data generator methods.

```
1   <?php
2
3   namespace DJWeb\Framework\Enums;
4
```

```php
enum FakerMethod: string
{
    case NAME = 'name';
    case EMAIL = 'safeEmail';
    case PHONE = 'phoneNumber';
    case DATE = 'date';
    case TIME = 'time';
    case ADDRESS = 'address';
    case COMPANY = 'company';
    case CITY = 'city';
    case STATE = 'state';
    case COUNTRY = 'country';
    case PASSWORD = 'password';
}
```

Listing 10.22: FakerMethod enum

```php
<?php

namespace DJWeb\Framework\DBAL\Models\Attributes;

use Attribute;
use DJWeb\Framework\Enums\FakerMethod;

#[Attribute(Attribute::TARGET_PROPERTY)]
readonly class FakeAs
{
    public function __construct(public FakerMethod $method
        )
    {}
}
```

Listing 10.23: FakeAs attribute

By following the SOLID approach, we can create a class with a complexity below 5 that automatically generates factories based on the model, maintaining transparency and simplicity of implementation. The use of enums allows us

to define the relationship between model fields and Faker generator methods, automating the data filling process.

```php
<?php

declare(strict_types=1);

namespace DJWeb\Framework\Console\Commands;

use DJWeb\Framework\Console\Attributes\AsCommand;
use DJWeb\Framework\Container\Contracts\ContainerContract;
use DJWeb\Framework\DBAL\Models\Attributes\FakeAs;
use ReflectionClass;
use ReflectionProperty;

#[AsCommand(name: 'make:factory')]
class MakeFactory extends MakeCommand
{
    protected $fakerClass;

    public function __construct(ContainerContract
       $container)
    {
        parent::__construct($container);
        $class = $container->getBinding('app.faker_class')
            ?? FakeAs::class;
        $this->fakerClass = $class;
    }
    protected function getStub(): string
    {
        $dir = dirname(__DIR__, 3);

        return $dir . '/stubs/factory.stub';
    }

    protected function getDefaultNamespace(): string
    {
        return $this->rootNamespace() . 'Database\\
           Factories';
    }
```

```
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: MakeFactory command

The template for generating factories looks as follows:

```
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: Factory template

10.10 Adding Seeders

Finally, we will add a seeder mechanism. It is an excellent complement to the database management system, as it allows for quickly and easily filling the database with test or initial data. It consists of three simple components: an abstract Seeder class, which defines the structure of seeders, and two commands: MakeSeeder and DatabaseSeeder.

10.10.1 How It Works

The abstract Seeder class has one abstract method `run`, which every inheriting class must implement. It is in this method that we define the logic for adding data to the database. The `call` method allows running another seeder, which enables nesting seeders and executing them in the proper order.

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace DJWeb\Framework\DBAL\Models;
6
7  abstract class Seeder
```

```
{
    abstract public function run(): void;

    protected function call(string $seeder): void
    {
        $instance = new $seeder();
        $instance->run();
    }
}
```

Listing 10.26: Seeder class

10.10.2 MakeSeeder Command

The `MakeSeeder` command is responsible for generating seeder files based on prepared templates (stubs). It allows for quickly creating new seeders without having to manually write the file from scratch.

```
<?php

declare(strict_types=1);

namespace DJWeb\Framework\Console\Commands;

use DJWeb\Framework\Console\Attributes\AsCommand;

#[AsCommand(name: 'make:seeder')]
class MakeSeeder extends MakeCommand
{
    protected function getStub(): string
    {
        $dir = dirname(__DIR__, 3);

        return $dir . '/stubs/seeder.stub';
    }

    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: MakeSeeder command

10.10.3 DatabaseSeed Command

The `DatabaseSeeder` command is responsible for executing the `run` function for a specified seeder class provided as a parameter. This makes it easy to fill the database using a specific seeder.

```
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: DatabaseSeed command

10.11 Verification of Correct Operation

In this section, we will show a few example tests for model handling.

10.11.1 Example Models

```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: Comment model

```php
<?php
```

```
2
3  namespace Tests\Helpers\Models;
4
5  use DJWeb\Framework\DBAL\Models\Attributes\HasMany;
6  use DJWeb\Framework\DBAL\Models\Attributes\HasManyThrough;
7  use DJWeb\Framework\DBAL\Models\Model;
8
9  class Company extends Model
10 {
11     public string $table {
12         get => 'companies';
13     }
14
15     #[HasMany(Post::class, foreign_key: 'company_id',
          local_key: 'id')]
16     public array $posts {
17         get => $this->relations->getRelation('posts');
18     }
19
20     #[HasManyThrough(Comment::class, Post::class, '
          company_id', 'post_id', 'id', 'id')]
21     public array $comments {
22         get => $this->relations->getRelation('comments');
23     }
24 }
```

Listing 10.30: Company model

```
1  <?php
2
3  namespace Tests\Helpers\Models;
4
5  use DJWeb\Framework\DBAL\Models\Model;
6
7  class CompanyUser extends Model
8  {
9
10     public string $table {
11         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: Company User model

```
<?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;
```

```
19          set {
20              $this->status = $value;
21              $this->markPropertyAsChanged('status');
22          }
23      }
24      #[FakeAs(FakerMethod::NAME)]
25      public string $name {
26          get => $this->name;
27          set {
28              $this->name = $value;
29              $this->markPropertyAsChanged('name');
30          }
31      }
32
33      public int $company_id {
34          get => $this->company_id;
35          set {
36              $this->company_id = $value;
37              $this->markPropertyAsChanged('company_id');
38          }
39      }
40
41      #[BelongsTo(Company::class, foreign_key: 'company_id',
            local_key: 'id')]
42      public Company $company {
43          get {
44              /** @var Company $model */
45              $model = $this->relations->getRelation('
                    company');
46              return $model;
47          }
48      }
49
50      protected array $casts = [
51          'published_at' => 'datetime',
52          'status' => Status::class,
53      ];
54  }
```

Listing 10.32: Post model

```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: User model

10.11.2 Basic Model Tests

In this section, we will check the correctness of basic operations performed on models.

```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: Basic model tests

10.11.3 Factory Tests

```
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 [
```

```
                    'name' => 'test',
                    'status' => 'draft',
                ];
            }

            protected function getModelClass(): string
            {
                return Post::class;
            }
        };

        $app = Application::getInstance();
        $builder = $this->createMock(
            InsertQueryBuilderContract::class);
        $any = $this->any();
        $stmt = $this->createMock(PDOStatement::class);
        $builder->expects($any)->method('table')->
            willReturnSelf();
        $builder->expects($any)->method('values')->
            willReturnSelf();
        $builder->expects($any)->method('execute')->
            willReturn($stmt);
        $builder->expects($any)->method('getInsertId')->
            willReturn('1');
        $app->set(InsertQueryBuilderContract::class,
            $builder);

        $factory = new $class();
        $post = $factory->create();
        $this->assertInstanceOf(Post::class, $post);
        $this->assertEquals(Status::draft, $post->status);

        $posts = $factory->createMany(2);
        $this->assertCount(2, $posts);
        $this->assertEquals(Status::draft, $posts[0]->
            status);
        $this->assertEquals(Status::draft, $posts[1]->
            status);
    }
}
```

Listing 10.35: Factory tests

10.11.4 Relationship Tests

Below is a test for the `HasMany` relationship; tests for the other relationships are analogous.

```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,
```

```
            new SelectQueryBuilder($this->mockConnection)
        );
        $this->queryBuilder = new QueryBuilder();
    }
}
```

Listing 10.36: HasMany relationship tests

10.11.5 Seeder Tests

```
<?php

namespace Tests\Console\Commands;

use DJWeb\Framework\Console\Application;
use DJWeb\Framework\Console\Output\Contacts\OutputContract
    ;
use DJWeb\Framework\Container\Contracts\ContainerContract;
use Tests\BaseTestCase;

class MakeSeederTest extends BaseTestCase
{
    private Application $app;
    private OutputContract $output;
    public function testMakeSeeder()
    {
        $seederName = 'UserSeeder';
        $file = 'UserSeeder.php';
        $_SERVER['argv'] = ['console/bin', 'make:seeder',
            $seederName];

        $this->app->set(OutputContract::class, $this->
            output);

        $this->output->expects($this->once())
            ->method('info')
            ->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: Seeder tests

10.11.6 Manual Tests

We will repeat the process, this time executing the testing procedure presented at the beginning of Chapter 7.

Important: The target model and migration for the `users` table will be shown later in the book in the chapter on authorization.

Creating a migration for the `users` table:

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

Then, we need to complete this migration according to the planned database structure.

```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: users table migration

The second step is to fully **automatically** generate the model. The only thing we need to do manually is to set the attributes that will supply our data factory.

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

Make sure that the generated model inherits from our base `User` model, which will ensure automatic password hashing.

```
<?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';
    }

    #[FakeAs(FakerMethod::NAME)]
    public string $name {
        get => $this->name;
        set {
            $this->name = $value;
            $this->markPropertyAsChanged('name');
        }
    }
    #[FakeAs(FakerMethod::EMAIL)]
    public string $email {
        get => $this->email;
        set {
            $this->email = $value;
            $this->markPropertyAsChanged('email');
        }
    }
    #[FakeAs(FakerMethod::DATE)]
    public Carbon $created_at {
        get => $this->created_at;
        set {
            $this->created_at = $value;
            $this->markPropertyAsChanged('created_at');
        }
    }
    #[FakeAs(FakerMethod::DATE)]
    public Carbon $updated_at {
        get => $this->updated_at;
        set {
            $this->updated_at = $value;
            $this->markPropertyAsChanged('updated_at');
        }
    }

    protected array $casts = [
        'created_at' => 'datetime',
        'updated_at' => 'datetime'
    ];
}
```

Listing 10.39: User model

The `FakeAs` attribute used in the code above takes an enum parameter with defined methods for the `Faker/Generator` class. The model itself fully leverages the Property Hooks mechanism introduced in PHP 8.4, which allows tracking real fields without using magic methods __get and __set.

With the table and model in place, we can automatically create the factory:

```
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: User factory

We can adjust the generated code as needed.

The final step is to create the seeder and populate the database:

```
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: UserSeeder

Chapter 26

Comparison with Existing Frameworks

To conclude this book, we will focus on comparing the created solution with other popular frameworks across several key aspects of framework functionality.

26.1 Minimum PHP Version and Required Dependencies

26.1.1 Created Solution

The author assumed that we will not be restricted, and because, as mentioned repeatedly, we use almost all new features introduced in PHP 8.4, this version of the language has been set as the minimum requirement. Apart from PSR interfaces, the `composer.json` file includes only a few essential libraries:

- **Carbon** - A library for working with dates and times in PHP, offering an intuitive interface and advanced operations such as manipulation, formatting, and date differences.
- **Symfony Mailer** - A modern tool for handling email sending, supporting

various protocols (SMTP, Sendmail, etc.) and ensuring security and easy configuration.

- **phpdotenv** - A tool for managing application configuration via `.env` files, allowing for easy storage and loading of environment variables.
- **HTMLPurifier** - A secure library for filtering HTML that removes potentially dangerous code and ensures compliance with standards.
- **Faker** - A tool for generating realistic random test data, such as names, addresses, or phone numbers, useful in testing and prototyping.

26.1.2 CodeIgniter 4

CodeIgniter 4 requires a minimum of PHP 7.4, making it more compatible with slightly older environments. In terms of dependencies, it remains minimalist, providing ready-to-use tools and limiting itself to the essential libraries embedded in the framework core.

26.1.3 Yii 2

Yii 2 is an older framework based on PHP 5.4 (in earlier releases) and PHP 7.2 in the latest updates. It requires the installation of several additional components via Composer, such as ORM tools, authorization, and data validation.

26.1.4 Laravel 11

Laravel 11 requires PHP 8.2 as the minimum version, making it compatible with modern environments but potentially requiring updates in older projects. The dependencies in Laravel are extensive, including Eloquent ORM, Blade Template Engine, tools for queues, and built-in support for caching systems.

26.1.5 Symfony 7

Symfony 7 sets PHP 8.1 as the minimum version while providing a very modular structure. Symfony's dependencies are extensive, as it relies on its own packages, such as Symfony Components, allowing the user to install only the necessary modules.

26.2 Code Complexity

The average code complexity in the framework is only 1.3, indicating high readability and simplicity in implementation. No class exceeded a complexity of 10, and only a few classes had a complexity greater than 5.

26.2.1 CodeIgniter 4

CodeIgniter focuses on simplicity and low code complexity. Due to the limited number of functions and classes, it is one of the easiest frameworks to learn, which contributes to its low average complexity.

26.2.2 Yii 2

Yii 2, due to its modular nature, allows for moderate code complexity, although integrating additional components can increase this value.

26.2.3 Laravel 11

Laravel is known for its rich functionality, which sometimes results in higher code complexity. The average code complexity in Laravel-based projects may be higher compared to minimalist frameworks.

26.2.4 Symfony 7

Symfony is flexible, but its code complexity depends on the number of components used. Fully developed applications may have a higher average complexity compared to other frameworks.

26.3 Automatic Dependencies

In Chapter 3, we implemented a container that is PSR-11 compliant, providing automatic binding of constructor and method parameters. The base class `Application` extends the container, enabling the setting of any dependencies. Following the Dependency Inversion Principle (DIP) from SOLID, elements can be resolved by both name and type, increasing flexibility in use. Additionally, using interfaces enables efficient testing of individual modules and facilitates application development, maintaining clarity and modularity of the code.

```php
<?php

use DJWeb\Framework\DBAL\Connection\MySqlConnection;
use DJWeb\Framework\DBAL\Contracts\ConnectionContract;
use DJWeb\Framework\DBAL\Models\Entities\DatabaseLog;


$app = \DJWeb\Framework\Web\Application::getInstance();

$app->set(ConnectionContract::class, new MySqlConnection());
/** @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 does not have built-in support for a full-fledged PSR-11-compliant dependency container, though it offers a simple service mechanism (Services) that allows for manually defining and managing dependencies in the application.

```php
<?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 provides a dependency injection mechanism through special configurations in arrays and enables the use of a dependency container. However, automatic binding of parameters in methods and constructors requires additional configuration.

```php
<?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 has a highly developed dependency container that allows for automatic binding of constructor and method parameters. Thanks to its convention-over-configuration approach, the framework automatically recognizes dependencies, and its integration with the provider system makes it one of the most advanced in this area.

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

```
3
4  use Yii;
5  use app\models\User;
6
7  class UserController extends \yii\web\Controller
8  {
9      private $user;
10
11     public function __construct($id, $module, User $user,
           $config = [])
12     {
13         $this->user = $user;
14         parent::__construct($id, $module, $config);
15     }
16
17     public function actionIndex()
18     {
19         $users = $this->user->find()->all();
20         return $this->render('index', ['users' => $users])
               ;
21     }
22 }
```

Listing 26.4: DI - Yii2

26.3.4 Symfony 7

Symfony 7 provides a full-fledged dependency container, PSR-11 compliant, with built-in support for automatic parameter binding and dependency resolution based on types. This solution is both flexible and efficient, allowing for precise control over service configuration.

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

```
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 and Middleware

In Chapter 4, a basic routing system was created, handling the simplest routes. In Chapter 12, middleware support fully compliant with PSR-15 was added. In Chapter 13, the router was enhanced with:

- Automatic model binding to controllers,
- Defining routes using controller attributes,
- Grouping routes.

Chapter 14 completed the router's development by adding form validation based on attributes and automatic binding of validators to controllers, similar to the spatie/laravel-data package.

```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 offers a basic routing system, allowing route definitions in configuration files using arrays. Middleware in CodeIgniter 4 is handled through filters, but it is not PSR-15 compliant and may require additional implementation for more advanced use cases.

```
<?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 26.7: routing - own framework

26.4.2 Yii 2

Yii 2 features a built-in routing system that integrates with the framework using configuration arrays or rules in configuration files. Middleware is not natively supported but can be implemented using additional components or extensions.

```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);
    }
}
```

```
34
35 // Apply Middleware
36 function behaviors()
37 {
38     return [
39         'auth' => [
40             'class' => AuthMiddleware::class,
41         ],
42     ];
43 }
```

Listing 26.8: routing - Yii 2

26.4.3 Laravel 11

Laravel 11 offers an advanced routing system, with routes typically defined in `routes/web` and `routes/api` files. Middleware is an integral part of the framework, allowing for advanced request and response handling in the application.

```
1  <?php
2  // Defining Routes
3  use App\Http\Controllers\UserController;
4
5  Route::get('/users', [UserController::class, 'index']);
6  Route::post('/users/create', [UserController::class, '
       create']);
7
8  // Middleware
9  namespace App\Http\Middleware;
10
11 use Closure;
12
13 class Authenticate
14 {
15     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 uses the `Routing` component, which allows routes to be defined in YAML, XML, PHP files, or using attributes. Middleware is implemented through EventListeners, which are fully PSR-15 compliant, providing flexibility and efficiency.

```
<?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 Database Management System

Due to limited space in this book, the author has chosen to implement a complete database management system, but only for the MySQL engine. This implementation enables:

- A system for managing database structure (Chapter 7),
- A modular *Query Builder* (Chapter 8) that allows for advanced querying (Chapter 8). Using design patterns such as Factory, Facade, and Decorator, this system is modular and adheres to SOLID principles,
- A migration system (Chapter 9) along with auxiliary commands `make:migration` and `migrate`,
- An ORM (Chapter 10), which is an intermediate solution between Doctrine and Active Record, fully utilizing the new features introduced in PHP 8.4. This enables the creation of a solution with full entity typing, resembling the Entity Framework known from C#. With the `make:model` command, model generation based on the database is almost automatic.

```php
<?php
//example migration
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\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;
```

```
53
54 class Company extends Model
55 {
56     public string $table {
57         get => 'companies';
58     }
59
60     public string $name {
61         get => $this->name;
62         set {
63             $this->name = $value;
64             $this->markPropertyAsChanged('name');
65         }
66     }
67
68
69     #[HasMany(Post::class, foreign_key: 'company_id',
       local_key: 'id')]
70     public array $posts {
71         get => $this->relations->getRelation('posts');
72     }
73
74     #[HasManyThrough(Comment::class, Post::class, '
       company_id', 'post_id', 'id', 'id')]
75     public array $comments {
76         get => $this->relations->getRelation('comments');
77     }
78 }
79
80 //example queries
81 User::query()->select()
82     ->where('id', '=', $userId)
83     ->whereGroup(function (WhereGroupCondition $group){
84         $group->where('activated', '=', 0)
85             ->orWhere('mail_confirmed', '=', 0)
86     })
87
88     ->first();
89 Session::query()->delete()->where('id', '=', $id)->delete
   ();
90 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 offers a built-in *Query Builder* that allows for concise and simple SQL queries. However, it lacks a full-fledged ORM, which might require developers to manually handle relationships between tables. The migration system is available, but its functionality is more limited compared to modern frameworks.

```
<?php

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

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

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

// Migration
```

```php
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');
    }
}
```

Listing 26.12: DBAL - CI4

26.5.2 Yii 2

Yii 2 uses Active Record as the primary method for handling databases. It is a simpler solution than Doctrine but less flexible in more complex cases. The migration system in Yii 2 is well integrated with the framework, and the *Query Builder* enables advanced query generation.

```
<?php

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

// Insert data
$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');
    }
}
```

Listing 26.13: DBAL - CI4

26.5.3 Laravel 11

Laravel uses Eloquent ORM, which combines the simplicity of Active Record with great flexibility, especially in handling relationships. It has an advanced *Query Builder* and a well-designed migration system, enabling easy database structure management and code generation via CLI commands such as `make:migration`.

```
<?php


// Eloquent ORM
use App\Models\User;

// Insert data
$user = new User();
$user->name = 'John Doe';
```

```php
$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 typically uses Doctrine ORM, which offers immense flexibility and the ability to work with different databases. Doctrine is more complex and requires configuration, but it provides powerful object-relational mapping capabilities as well as support for migrations and schema management. Symfony also supports *Query Builder* as part of Doctrine.

```
<?php


// Eloquent ORM
use App\Models\User;

// Insert data
$user = new User();
$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 26.15: DBAL - Laravel 11

26.6 Console Applications

A modern framework is not only for applications running in the browser but also for fully functional console applications. In Chapter 6, a base command class was created, using attributes to pass parameters and automatically register commands. The input and output of commands are handled using streams compliant with PSR-7. A base command `MakeCommand` was created, enabling the creation of other commands for automatically generating framework elements, such as models, migrations, controllers, etc.

```
<?php

use DJWeb\Framework\Console\Attributes\AsCommand;
use DJWeb\Framework\Console\Attributes\CommandArgument;
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 has a console system with limited functionality compared to modern frameworks. Commands can be defined manually, but they are not automatically registered and do not use more advanced solutions like attributes or PSR-7.

```
<?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 supports console applications via the `yii\console\Controller` component. Commands are defined as classes of console controllers, and parameters can be passed as CLI arguments. However, it lacks automatic registration and the more modern approach introduced by 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 has an advanced console application system based on the Artisan package. Commands are defined using classes that inherit from `Illuminate Console Command`. Parameters can be easily defined within the classes, and the automatic registration system and support for code generation using commands such as `make:model` or `make:controller` make Artisan one of the most advanced solutions.

```php
<?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 uses the `Console` component, one of the most versatile solutions for building console applications. Commands are defined as classes with full support for dependency injection container services and automatic parameter registration. This component is also fully compatible with PSR-7.

```php
<?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');
    }

    protected function execute(InputInterface $input,
       OutputInterface $output): int
    {
        $name = $input->getArgument('name');
        $output->writeln("Hello, $name!");
        return Command::SUCCESS;
    }
}
```

Listing 26.20: Console - Symfony 7

26.7 Views

The view rendering system created in this book is highly versatile and allows integrating any existing view rendering mechanism. Using this interface, Twig was integrated in Chapter 15, a custom Blade adapter was created in Chapter 16, and in Chapter 17, a modern web application was implemented with a custom Inertia.js adapter based on Blade files combined with Vue 3.

```php
<?php

use DJWeb\Framework\Routing\Attributes\Route;
use DJWeb\Framework\Routing\Attributes\RouteGroup;
use DJWeb\Framework\Routing\Controller;
use DJWeb\Framework\View\Inertia\Inertia;

#[RouteGroup('views')]
class ControllerRenderingBlade extends Controller
{
    #[Route('blade')]
    public function blade(): ResponseInterface
    {
        $this->withRenderer('blade');
        return $this->render('index.blade.php', ['user' =>
            'test', 'x' => 3]);
    }

    #[Route('twig')]
    public function twig(): ResponseInterface
    {
        $this->withRenderer('twig');
        return $this->render('index.twig', ['user' => '
           test']);
    }

    #[Route('/inertia', 'GET')]
    public function index(): ResponseInterface
    {
        return Inertia::render('InertiaRendering.vue', [
            'test' => 'test'
```

```
30            ]);
31      }
32
33  }
```

Listing 26.21: View - own framework

26.7.1 CodeIgniter 4

CodeIgniter 4 offers a simple view system based on standard PHP files, without support for advanced templates like Blade or Twig. The lack of built-in support for modern tools like Inertia.js makes it less flexible compared to contemporary requirements.

```
1  <?php
2  // Controller
3  namespace App\Controllers;
4
5  class Home extends BaseController
6  {
7      public function index()
8      {
9          return view('welcome_message', ['title' => '
               Welcome to CodeIgniter!']);
10     }
11 }
12
13 // View (app/Views/welcome_message.php)
14 <!DOCTYPE html>
15 <html>
16 <head>
17     <title><?= esc($title) ?></title>
18 </head>
19 <body>
20 <h1><?= esc($title) ?></h1>
21 <p>Hello, CodeIgniter!</p>
```

```
</body>
</html>
```

Listing 26.22: View - CI4

26.7.2 Yii 2

Yii 2 uses a view system based on PHP files, with optional support for Twig templates. The framework also allows for extending the view mechanism, enabling integration with other template systems, but this requires additional configuration.

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

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 26.23: View - CI4

26.7.3 Laravel 11

Laravel 11 uses Blade templates, which are fast, lightweight, and very flexible. The framework also supports Inertia.js, making it an ideal choice for modern web applications, combining PHP backend with frontend based on Vue 3, React, or Svelte.

```
<?php


// 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>
```

```
26
27 // Example with Inertia.js
28 return Inertia::render('Home', ['title' => 'Welcome to
      Laravel with Inertia!']);
29 \end{verbatim}
```

Listing 26.24: View - Laravel 11

26.7.4 Symfony 7

Symfony 7 uses Twig by default as the template engine, offering advanced view rendering capabilities. Thanks to its modular structure, Symfony allows integration with other mechanisms, but it does not have native support for systems like Inertia.js, which would need to be manually implemented.

```
1  <?php
2  // Controller
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\
       AbstractController;
6  use Symfony\Component\HttpFoundation\Response;
7
8  class HomeController extends AbstractController
9  {
10     public function index(): Response
11     {
12         return $this->render('home/index.html.twig', [
13             'title' => 'Welcome to Symfony!',
14         ]);
15     }
16 }
17
18 // Twig Template (templates/home/index.html.twig)
19 <!DOCTYPE html>
20 <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 Exception Handling

The exception handling system created in this book (Chapter 18) consists of three main components:

- Using base PHP functions for registering error and exception handlers,
- A browser-based interface based on Vue 3, allowing easy debugging and error tracking in developer mode,
- A console interface that is a full-fledged console application, allowing debugging of exceptions in the console, including browsing `backtrace`.

26.8.1 CodeIgniter 4

CodeIgniter 4 provides a simple error handling mechanism that generates HTML reports for development mode and logs for production mode. It lacks more advanced tools such as a console debugger or frontend-based visual interfaces.

26.8.2 Yii 2

Yii 2 offers a built-in exception handling system with detailed error reports available in development mode, including views showing the details of `backtrace`. However, it lacks native support for frontend-based interfaces like Vue 3.

26.8.3 Laravel 11

Laravel 11 uses `Ignition`, an advanced exception handling tool that provides readable error reports with a browser-based interface. However, it does not include a built-in console error debugger, meaning that debugging in the CLI requires additional tools.

26.8.4 Symfony 7

Symfony 7 includes an advanced error handling system that generates detailed reports in development mode, including visual summaries of exceptions in the browser. The integration with the `Debug` component allows for in-depth `backtrace` analysis, though a console debugger is not part of the standard toolset.

26.9 Session and Cookies

Handlers for sessions based on databases and files have been created, demonstrating the advantages and disadvantages of both solutions. Both solutions utilize modern data encryption based on the `sodium` algorithm available since PHP 7.2. Data stored in cookies is also encrypted. Additionally, thanks to the `key:generate` command, it is possible to generate an encryption key and save it in the `.env` file.

26.9.1 CodeIgniter 4

CodeIgniter 4 supports sessions using files, databases, or cache memory, with an option for encrypting session data. Cookie handling is simpler, with no native support for advanced encryption, which requires additional configuration or libraries.

26.9.2 Yii 2

Yii 2 offers support for sessions and cookies with the ability to store data in files, databases, or cache memory. Data in cookies can be encrypted using built-in tools, but this mechanism requires manual configuration of encryption keys.

26.9.3 Laravel 11

Laravel 11 supports sessions stored in files, databases, cache memory, Redis, and other systems. Cookies are encrypted by default using a key generated by the `key:generate` command, ensuring high security. The framework also provides an easy way to manage encryption keys in the `.env` file.

26.9.4 Symfony 7

Symfony 7 supports sessions stored in files, databases, or cache memory, with an option for encryption. Cookie handling in Symfony requires manual configuration of encryption using the `Security` component, which provides great flexibility but may be more complicated to configure.

26.10 Email Sending

The system created in the book wraps the low-level `symfony/mailer` library according to SOLID principles, also enabling HTML email sending based

on Blade or Twig templates. This solution provides flexibility and modularity, allowing easy adaptation to the needs of the application.

```php
<?php

//php console/bin WelcomeMailable

class WelcomeMailable extends Mailable
{
    public function __construct(
        private User $user
    ) {}

    public function content(): Content
    {
        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 has a built-in `Email` class that handles email sending via SMTP, Mail, and Sendmail protocols. However, it lacks support for modern HTML templates, which requires additional integrations.

```php
<?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 supports email sending via the `yii\mailer` component, which allows using various adapters such as SwiftMailer. HTML email support is possible, but integration with Twig or Blade templates requires additional configuration.

```php
<?php


```

```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 offers an advanced email sending system based on the `Illuminate\Mail` class, which supports various message delivery systems such as SMTP, Mailgun, and AWS SES. Laravel natively supports Blade templates for generating HTML email content.

```php
<?php

use Illuminate\Support\Facades\Mail;

// Sending a simple email
```

```php
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 uses the `Mailer` component, which provides high flexibility and support for various email sending protocols. With integration with the `Twig` component, advanced HTML templates can be created, but there is no native support for other template engines like Blade.

```php
<?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 Authorization

The created authorization system is quite simple – it is based on the `User` class and a basic role and permission system. Example views for registration and password recovery were created in Vue 3, providing a modern user interface.

26.11.1 CodeIgniter 4

CodeIgniter 4 does not have a built-in authorization system but offers tools like `Filters`, which can be used to implement a role and permission system. Many features, however, require additional libraries or manual implementation.

26.11.2 Yii 2

Yii 2 offers an `RBAC` (Role-Based Access Control) system that allows managing roles and permissions. The system is flexible, but its configuration may be more complex compared to simpler frameworks.

26.11.3 Laravel 11

Laravel 11 features a built-in authorization system based on `Policies` and `Gates`, allowing for detailed control over access to resources. It also supports role and permission systems through additional packages and provides ready-to-use templates for user registration and management.

26.11.4 Symfony 7

Symfony 7 uses the `Security` component, which allows for advanced access control based on roles and special rules. This system is highly flexible but may require significant configuration effort, especially for simpler use cases.

26.12 Cache

The caching system is fully compliant with PSR-6. A storage solution based on plain files and Redis has been created. Additionally, memory management strategies and functions like `Cache::remember` have been implemented, allowing any callback to be cached, greatly simplifying working with cache data.

26.12.1 CodeIgniter 4

CodeIgniter 4 offers a simple cache system supporting files, Redis, Memcached, and other popular caching systems. However, it lacks PSR-6 compliance, which limits the portability of code between projects.

26.12.2 Yii 2

Yii 2 includes a built-in caching system supporting various mechanisms such as files, Redis, and Memcached. While the system is efficient and flexible, it is not PSR-6 compliant, which could make integration with external libraries difficult.

26.12.3 Laravel 11

Laravel 11 offers an advanced caching system supporting files, Redis, Memcached, and other mechanisms. Features like `Cache::remember` are natively supported, making the system developer-friendly. Laravel does not natively support PSR-6 but can be extended to achieve this.

26.12.4 Symfony 7

Symfony 7 uses the `Cache` component, which is fully compliant with PSR-6 and PSR-16. It supports various caching storage systems and advanced memory management mechanisms, making it one of the most flexible solutions.

26.13 Event System

The event handling system created in the book is a simple implementation of PSR-14, ensuring compliance with modern PHP standards. The core of the system is the `EventManager` class, which acts as a facade, simplifying event management. Events and their listeners are defined in the configuration file, making it easier to configure and extend the application.

```
<?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 26.31: Mail - Symfony 7

26.13.1 CodeIgniter 4

CodeIgniter 4 has a simple event system that allows for registering and triggering listeners. The system is intuitive, but the lack of PSR-14 compliance may limit its use in more advanced projects.

```
<?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 offers a built-in event system that allows registering handlers and integrates with the application lifecycle. The system is not PSR-14 compliant and requires event registration directly in the code, which can hinder scalability.

```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 26.33: Events - CI4

26.13.3 Laravel 11

Laravel 11 uses an event system that allows defining listeners and events in dedicated classes. While this system does not implement PSR-14, it is very flexible and well-integrated with other parts of the framework, such as 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 uses the `EventDispatcher` component, which is compliant with PSR-14 and offers advanced event handling. Listeners can be registered in configuration files, making this system highly flexible and aligned with modern standards.

```
<?php

// Event class
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 Queues

The mechanism created in this book implements both job handling and the scheduler. Jobs inherit from the abstract `Job` class, which, thanks to the use of the `#[SerializeTo]` attribute, allows full control over automatic serialization. The system supports Redis and database-based backends, ensuring flexibility based on project requirements.

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

Listing 26.36: Jobs - own framework

26.14.1 CodeIgniter 4

CodeIgniter 4 does not have a built-in queue mechanism, requiring the use of external libraries or manual implementation of queue handling. The lack of support for popular backends like Redis or databases is a limitation for more advanced projects.

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

Listing 26.37: Jobs - CI4

26.14.2 Yii 2

Yii 2 offers a queue system as an extension, supporting various backends such as Redis, RabbitMQ, or databases. Jobs can be defined as classes, but the lack of native serialization support requires additional configuration or the use of external libraries.

```php
<?php

// Define a job
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 has an advanced queue system that supports multiple backends such as Redis, AWS SQS, RabbitMQ, and databases. Jobs are defined as classes inheriting from `Illuminate\Queue\Job`, and the framework natively handles their serialization and integration with the scheduler system.

```
<?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 uses the `Messenger` component, which provides queue handling and message sending functionality. The mechanism supports various backends, including Redis and databases, and allows full control over serialization and deserialization of jobs. However, the scheduler requires separate configuration or additional tools.

```
<?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
    {
```

```
16          // Logic for sending email
17      }
18  }
```

Listing 26.40: Jobs - Symfony 7

26.15 WebSocket

The custom WebSocket server implemented within the framework is based on the low-level `react/event-loop` library. The server allows for real-time bidirectional communication between the client and the server. It can be started using the **ws:run** command. This solution integrates seamlessly with the previously described frontend built using Vue 3, enabling the creation of modern real-time applications.

26.15.1 CodeIgniter 4

CodeIgniter 4 does not have built-in support for WebSocket. Implementing this technology requires manual integration with external libraries like Ratchet or ReactPHP, which can be time-consuming.

26.15.2 Yii 2

Yii 2 also does not provide native WebSocket support. However, it is possible to use additional extensions, such as `yiisocket`, which allow implementing WebSocket servers, though this requires more work.

26.15.3 Laravel 11

Laravel 11 uses the Reverb technology, which enables real-time communication between the server and client. However, Reverb is limited to server-

to-client communication, meaning it does not support full-fledged bidirectional WebSocket communication. For more advanced WebSocket use cases, additional packages like `laravel-websockets` are required.

26.15.4 Symfony 7

Symfony 7 does not provide native WebSocket support, but components like `Ratchet` or integrations with ReactPHP can be used to implement WebSocket servers. This approach, however, requires manual configuration and additional resources.

Index