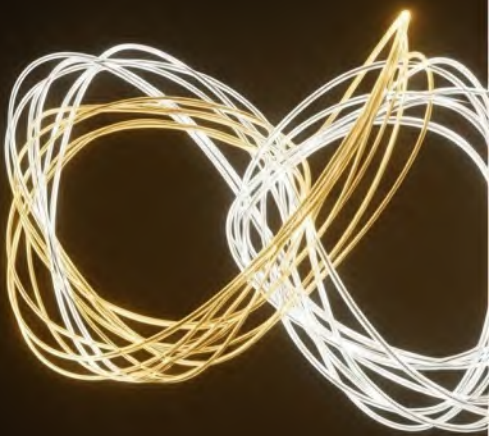
LARAVEL MVC WEB PROGRAMMING: PROFESSIONAL (INTEGRATIVE-GENERATIVE AI EDITION)



KEY CONTENTS:

Student Price Book Center

คำนำ

ในโลกของการพัฒนาเว็บแอปพลิเคชันยุคใหม่ การสร้างระบบที่ไม่เพียงแต่ “ใช้งานได้” แต่ยังต้อง “ปรับขยายได้ง่าย ดูแลรักษาได้ดี และปลอดภัย” ได้กลายมาเป็นมาตรฐานใหม่ของอุตสาหกรรม นักพัฒนายุคปัจจุบันจึงต้องก้าวข้ามจากการเขียนโค้ดเชิงเทคนิคแบบพื้นฐาน ไปสู่การออกแบบสถาปัตยกรรมซอฟต์แวร์ที่มีความยั่งยืน มีระเบียบ และรองรับการเติบโตในระดับองค์กรได้จริง หนังสือ *Laravel MVC Web Programming: Professional* เล่มนี้ถูกสร้างขึ้นเพื่อตอบโจทย์นั้นอย่างแท้จริง

ไม่เพียงแค่พาผู้อ่านให้เข้าใจวิธีใช้ Laravel อย่างมืออาชีพ แต่ยังชี้ให้เห็นถึงแนวคิดเบื้องหลังที่ทำให้ Laravel สามารถรองรับระบบซับซ้อนในระดับองค์กรได้อย่างมั่นคง ไม่ว่าจะเป็น Design Patterns, Clean Architecture, การทดสอบระบบอย่างครอบคลุม, การเตรียมระบบสำหรับ Production, หรือแม้กระทั่งการพัฒนา Microservices ที่เชื่อมต่อกันอย่างมีประสิทธิภาพ

ในบทที่ 13 ผู้อ่านจะได้รู้จักกับ *Design Patterns และ Clean Architecture ใน Laravel* (หน้า 1) ซึ่งถือเป็นแกนหลักของการออกแบบระบบที่ดี เราจะสำรวจ Pattern ที่นำไปใช้ได้จริงใน Laravel เช่น Repository, Strategy, Factory, รวมถึงแนวคิด Clean Architecture ที่แยก Layer อย่างชัดเจน (เช่น Controller ⟶ Service ⟶ Domain ⟶ Infrastructure) เพื่อให้ระบบมีความเป็นอิสระของแต่ละชั้นและพร้อมต่อการทดสอบหรือปรับเปลี่ยนในอนาคต

บทที่ 14 ว่าด้วยเรื่อง *การทดสอบด้วย PHPUnit และ Laravel Testing* (หน้า 105) ผู้อ่านจะได้ลงมือเขียน Unit Test สำหรับ Model, Controller และ Service ที่ทำงานเบื้องหลัง พร้อมทั้งเรียนรู้ Feature Test สำหรับตรวจสอบระบบในมุมมองของผู้ใช้ การ Mock ข้อมูลเพื่อแยกการทดสอบแต่ละชั้น การใช้ Test Doubles รวมถึงการตั้งค่า Continuous Integration (CI) เพื่อให้การทดสอบเป็นอัตโนมัติและต่อเนื่อง เหมาะสำหรับการทำงานเป็นทีมและในระบบที่มีการ deploy บ่อย

เมื่อระบบพร้อมใช้งานจริง การเตรียมระบบ Production คือสิ่งที่ขาดไม่ได้ บทที่ 15 *การ Deploy Laravel Application สู่ Production* (หน้า 190) จะพาผู้อ่านตั้งแต่การตั้งค่า Environment การใช้ Laravel Forge หรือการ Deploy แบบ Manual ไปจนถึงการตั้งค่า Web Server และ Queue Worker สำหรับงานเบื้องหลัง พร้อมเทคนิคการทำ Monitoring และ Logging เพื่อให้ระบบทำงานได้เสถียร ปลอดภัย และสามารถติดตามปัญหาได้อย่างรวดเร็ว

สุดท้าย บทที่ 16 *การพัฒนา Microservices และ Advanced API* (หน้า 249) จะเปิดโลกการออกแบบระบบแบบแยกส่วนด้วยแนวคิด Microservices โดยใช้ Laravel เป็นฐานในการสร้างบริการแต่ละตัว พร้อมทั้งเรียนรู้การออกแบบ RESTful และ GraphQL API การจัดการ Authentication แบบ OAuth2 และ JWT การสื่อสารระหว่าง Services ทั้งแบบ HTTP, gRPC หรือผ่าน Message Queue และการนำ Docker มาใช้ในการพัฒนาและ Deploy อย่างมีระบบ พร้อมตัวอย่างจริงที่สามารถนำไปต่อยอดในองค์กรได้ทันที

หนังสือเล่มนี้เหมาะสำหรับนักพัฒนาที่ผ่านขั้นพื้นฐาน Laravel มาแล้ว และกำลังมองหาวิธี “ยกระดับ” การพัฒนาไปสู่ระบบที่มั่นคง ปรับขยายง่าย และเป็นมืออาชีพอย่างแท้จริง ด้วยแนวทางการเขียนโค้ดเชิงสถาปัตยกรรม การทดสอบเชิงระบบ และการจัดการรอบด้านสำหรับระบบ Production และ Microservices หากผู้อ่านคือนักพัฒนาที่ต้องการสร้าง Laravel Application ที่ “พร้อมใช้จริงในองค์กร” หนังสือเล่มนี้คือคำตอบ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 13

Design Patterns และ Clean Architecture ใน Laravel

(Design Patterns and Clean Architecture in Laravel)

เนื้อหา

- Design Patterns และ Clean Architecture ใน Laravel
- Design Patterns และ Clean Architecture ใน Laravel (รายละเอียดเชิงลึก)
- การประยุกต์ใช้ Design Patterns ใน Laravel
- การออกแบบระบบแบบ Clean Architecture
- แนวคิดและรายละเอียดเชิงลึกของการแยก Layer แบบ Clean Architecture ใน Laravel

---

บทนำ

บทที่ 13: Design Patterns และ Clean Architecture ใน Laravel

การพัฒนาเว็บแอปพลิเคชันที่มีคุณภาพและยืดหยุ่นในระยะยาวต้องอาศัยแนวทางการออกแบบซอฟต์แวร์ที่ดี การประยุกต์ใช้ **Design Patterns** เช่น Singleton, Factory และ Strategy ช่วยแก้ไขปัญหาการออกแบบที่ซ้ำซ้อนและเพิ่มความสามารถในการบำรุงรักษาโค้ด ตัวอย่างเช่น Singleton ช่วยควบคุมการสร้างอินสแตนซ์ของคลาสให้มีเพียงตัวเดียว Factory ช่วยสร้างออบเจกต์โดยไม่ต้องผูกมัดกับคลาสเฉพาะ และ Strategy ช่วยให้การเปลี่ยนแปลงพฤติกรรมของโปรแกรมเป็นไปอย่างยืดหยุ่น

นอกจากนั้น แนวคิด **Clean Architecture** เป็นกรอบการออกแบบระบบที่เน้นการแยกส่วนประกอบต่างๆ ของแอปพลิเคชันออกจากกันอย่างชัดเจน เพื่อลดความซับซ้อนและทำให้โค้ดมีความเป็นระเบียบ อ่านง่าย และง่ายต่อการทดสอบ โดย Clean Architecture มุ่งเน้นการแยกความรับผิดชอบของแต่ละชั้นงานอย่างชัดเจน

การแยกระบบออกเป็น **Domain Layer**, **Application Layer** และ **Infrastructure Layer** เป็นหัวใจหลักของ Clean Architecture โดย Domain Layer จะเก็บตรรกะทางธุรกิจที่สำคัญ Application Layer จะประสานงานระหว่าง Domain กับส่วนอื่นๆ ของระบบ ส่วน Infrastructure Layer จะดูแลการเชื่อมต่อกับระบบภายนอก เช่น ฐานข้อมูล หรือบริการอื่นๆ การจัดโครงสร้างเช่นนี้ช่วยให้ระบบมีความยืดหยุ่นและสามารถปรับเปลี่ยนส่วนใดส่วนหนึ่งโดยไม่ส่งผลกระทบกับส่วนอื่น

บทนี้จึงเน้นให้นักพัฒนาทำความเข้าใจและนำ Design Patterns มาประยุกต์ใช้ร่วมกับแนวทาง Clean Architecture ในการออกแบบระบบ Laravel เพื่อให้ได้โครงสร้างโปรแกรมที่มีคุณภาพสูง มีความยืดหยุ่นและง่ายต่อการบำรุงรักษาในระยะยาว

Design Patterns และ Clean Architecture ใน Laravel

- การประยุกต์ใช้ Design Patterns เช่น Singleton, Factory, Strategy
- การออกแบบระบบแบบ Clean Architecture
- การแยก Domain Layer, Application Layer และ Infrastructure Layer

---

1. การประยุกต์ใช้ Design Patterns ใน Laravel

Laravel เป็น PHP Framework ที่ออกแบบมาให้ใช้แนวคิด OOP และ Design Patterns ได้ง่าย ตัวอย่าง Design Patterns ที่นิยมใน Laravel ได้แก่

- Singleton Pattern

ใช้เพื่อให้คลาสมี instance เดียวตลอดแอป เช่น Laravel Service Container เองก็ทำงานแบบ Singleton กับ Services หลายตัว

```
class Logger
{
    private static $instance;

    private function __construct() {}

    public static function getInstance()
    {
        if (!self::$instance) {
            self::$instance = new Logger();
        }
        return self::$instance;
    }

    public function log($message)
    {
        echo $message;
    }
}
```

ใน Laravel เรามักใช้ Service Container แทนการสร้าง Singleton เอง

- Factory Pattern

ใช้สร้าง Object หลายประเภทโดยไม่ต้องระบุคลาสที่ชัดเจน ช่วยแยกการสร้าง Object ออกจากการใช้งาน

ตัวอย่างการสร้าง Factory สำหรับการสร้าง Notification ต่าง ๆ

```
interface NotificationInterface {
    public function send($message);
}

class EmailNotification implements NotificationInterface {
    public function send($message) {
        // ส่งอีเมล
    }
}

class SmsNotification implements NotificationInterface {
    public function send($message) {
        // ส่ง SMS
    }
}

class NotificationFactory {
    public static function create($type): NotificationInterface {
        if ($type === 'email') {
            return new EmailNotification();
        }
        elseif ($type === 'sms') {
            return new SmsNotification();
        }
        throw new Exception("Notification type not supported.");
    }
}
```

การเรียกใช้

```
$notification = NotificationFactory::create('email');
$notification->send("Hello world");
```

- Strategy Pattern

ช่วยสลับพฤติกรรมของอ็อบเจกต์โดยไม่ต้องแก้ไขโค้ดหลัก ใช้ interface และ class หลายตัว

```
interface PaymentStrategy {
    public function pay($amount);
}

class CreditCardPayment implements PaymentStrategy {
    public function pay($amount) {
        // Logic ชำระเงินผ่านบัตรเครดิต
    }
}

class PaypalPayment implements PaymentStrategy {
    public function pay($amount) {
        // Logic ชำระเงินผ่าน Paypal
    }
}

class PaymentContext {
    private $strategy;

    public function __construct(PaymentStrategy $strategy) {
        $this->strategy = $strategy;
    }

    public function executePay($amount) {
        $this->strategy->pay($amount);
    }
}
```

การใช้

```
$payment = new PaymentContext(new PaypalPayment());
$payment->executePay(1000);
```

---

2. การออกแบบระบบแบบ Clean Architecture

Clean Architecture แนะนำให้แยกโค้ดออกเป็นชั้น (Layer) เพื่อให้ระบบยืดหยุ่น ดูแลง่าย และทดสอบง่าย

โครงสร้างหลัก 4 ชั้น

- **Entities (Domain Layer):** กฎและข้อมูลธุรกิจ (Business Logic) แบบนามธรรม ไม่ขึ้นกับ Framework หรือ Infrastructure ใด ๆ
- **Use Cases (Application Layer):** กำหนดการทำงานเฉพาะของระบบ (Business Use Cases) โดยใช้ Entities
- **Interface Adapters (Interface Layer):** ตัวแปลงข้อมูลและรับ/ส่งข้อมูลระหว่างระบบกับโลกภายนอก เช่น Controller, Presenter, Gateway
- **Frameworks & Drivers (Infrastructure Layer):** โค้ดที่เชื่อมต่อกับฐานข้อมูล, UI, network, Laravel Framework, Package ต่าง ๆ

---

3. การแยก Domain Layer, Application Layer และ Infrastructure Layer ใน Laravel

ใน Laravel ปกติเราใช้โครงสร้างแบบ MVC แต่ Clean Architecture แนะนำการแยกโฟลเดอร์และหน้าที่ชัดเจน เช่น

```
app/
├── Domain/              # Entities, Models, Business Logic
│   └── User.php
├── Application/         # Use Cases, Service Classes
│   └── UserService.php
├── Infrastructure/      # Database Repositories, External Services
│   └── Repositories/
│       └── UserRepository.php
├── Http/                # Controller, Request, Response
│   └── Controllers/
│       └── UserController.php
```

ตัวอย่างการแยกชั้น

Domain/User.php (Entity)

```
<?php
namespace App\Domain;

class User
{
```

```
    public $id;
    public $name;
    public $email;

    public function __construct($id, $name, $email)
    {
        $this->id = $id;
        $this->name = $name;
        $this->email = $email;
    }

    // Business logic เช่น validation
    public function canRegister()
    {
        return filter_var($this->email, FILTER_VALIDATE_EMAIL) !== false;
    }
}
```

Application/UserService.php (Use Case / Service Layer)

```
<?php
namespace App\Application;

use App\Domain\User;
use App\Infrastructure\Repositories\UserRepository;

class UserService
{
    private $userRepository;

    public function __construct(UserRepository $repo)
    {
        $this->userRepository = $repo;
    }

    public function registerUser($data)
```

```
    {
        $user = new User(null, $data['name'], $data['email']);

        if (!$user->canRegister()) {
            throw new \Exception("Invalid email");
        }

        return $this->userRepository->save($user);
    }
}
```

Infrastructure/Repositories/UserRepository.php

```
<?php
namespace App\Infrastructure\Repositories;

use App\Domain\User;
use App\Models\User as EloquentUser;

class UserRepository
{
    public function save(User $user)
    {
        $model = new EloquentUser();
        $model->name = $user->name;
        $model->email = $user->email;
        $model->save();

        $user->id = $model->id;

        return $user;
    }
}
```

Http/Controllers/UserController.php

```
<?php
namespace App\Http\Controllers;
```

```
use App\Application\UserService;
use Illuminate\Http\Request;

class UserController extends Controller
{
    private $userService;

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

    public function register(Request $request)
    {
        try {
            $user = $this->userService->registerUser($request->only(['name', 'email']));
            return response()->json(['success' => true, 'user' => $user]);
        } catch (\Exception $e) {
            return response()->json(['success' => false, 'error' => $e->getMessage()], 400);
        }
    }
}
```

สรุป

ส่วน	หน้าที่หลัก	ตัวอย่างใน Laravel
Domain Layer	กฎและข้อมูลธุรกิจ (Entities)	App\Domain\User.php
Application Layer	ใช้งานธุรกิจจริง (Use Cases, Services)	App\Application\UserService.php
Infrastructure Layer	เชื่อมต่อฐานข้อมูล, API, External	App\Infrastructure\Repositories\UserRepository.php
Interface Layer	UI, Controller, Routes	App\Http\Controllers\UserController.php

บทที่ 13: Design Patterns และ Clean Architecture ใน Laravel (รายละเอียดเชิงลึก)

1. ทำไมต้องใช้ Design Patterns และ Clean Architecture?

- **Maintainability (ง่ายต่อการดูแลรักษา)**
  โค้ดมีโครงสร้างชัดเจน แยกความรับผิดชอบ (Separation of Concerns) ทำให้แก้ไขและพัฒนาต่อได้ง่าย
- **Scalability (ขยายระบบได้ง่าย)**
  เมื่อระบบซับซ้อนขึ้น การแยกเลเยอร์และใช้ pattern จะช่วยให้เพิ่มฟีเจอร์ใหม่ ๆ โดยไม่กระทบส่วนอื่นมาก
- **Testability (ง่ายต่อการทดสอบ)**
  แยก business logic ออกจาก framework ช่วยให้เขียน unit test ได้สะดวก
- **Reusability (นำโค้ดกลับมาใช้ใหม่)**
  โค้ดที่ออกแบบดีจะนำกลับมาใช้ในหลายจุดได้โดยไม่ต้องเขียนซ้ำ

2. Design Patterns ที่นิยมใน Laravel

2.1 Singleton Pattern

- วัตถุประสงค์: สร้าง instance เดียวของคลาสตลอดอายุโปรแกรม
- ใน Laravel: Service Container ทำงานแบบ singleton กับ service หลายตัว เช่น Logger, Cache, Config

ประโยชน์: ลดการสร้าง object ซ้ำซ้อน ประหยัด resource

2.2 Factory Pattern

- วัตถุประสงค์: แยกส่วนการสร้าง object ออกจากการใช้งาน (Decoupling Object Creation)
- ตัวอย่าง: การสร้าง notification หลายชนิด โดย factory จะสร้างอ็อบเจ็กต์ชนิดที่ต้องการตามเงื่อนไข

ประโยชน์: ลด coupling, เพิ่มความยืดหยุ่นในการเพิ่มประเภทใหม่ ๆ

2.3 Strategy Pattern

- วัตถุประสงค์: เลือก algorithm หรือพฤติกรรมระหว่าง runtime โดยไม่ต้องแก้โค้ดหลัก
- ใน Laravel: ใช้เปลี่ยนวิธีทำงาน เช่น วิธีชำระเงินหลายแบบ, วิธีการแจ้งเตือนต่าง ๆ

ประโยชน์: เพิ่มความยืดหยุ่น, ลดการใช้ if/else ซ้อนกัน

3. Clean Architecture คืออะไร?

3.1 แนวคิดหลัก

- **แยกความรับผิดชอบของแต่ละส่วน** ให้อยู่ในชั้นที่เหมาะสม
- **Dependency Rule:** โค้ดจากชั้นใน (Domain, Business Logic) ห้ามขึ้นกับชั้นนอก (Framework, DB)
- **Isolation:** ชั้นในไม่ควรขึ้นกับเทคโนโลยีภายนอกเพื่อให้เปลี่ยนแปลงได้ง่าย

3.2 ชั้นต่าง ๆ ใน Clean Architecture

ชั้น	เนื้อหา/หน้าที่	ข้อดี/ตัวอย่าง
Entities (Domain Layer)	กฎธุรกิจ, Entities ที่เป็นนามธรรม	คลาส Model ธุรกิจ, Validation Logic
Use Cases (Application Layer)	กำหนดการทำงาน, Business Logic เฉพาะ use case	Service Classes, Interactors
Interface Adapters (Interface Layer)	ตัวแปลงข้อมูล, Controller, Presenter, Gateway	Controller, API Resource, Repository Interface
Frameworks & Drivers (Infrastructure Layer)	Implementation จริง เช่น DB, Framework, External APIs	Eloquent Model, Laravel DB, Third-party SDK

3.3 ความสัมพันธ์ของชั้น

- ชั้นในสามารถเรียกใช้ชั้นที่อยู่ภายในเท่านั้น
- ชั้นนอกขึ้นกับชั้นในได้ แต่ไม่กลับกัน
- ตัวอย่างเช่น Domain Layer ไม่ควรขึ้นกับ Laravel Framework แต่ Laravel Framework สามารถเรียกใช้ Domain Layer ได้

4. การนำ Clean Architecture มาใช้กับ Laravel

4.1 โครงสร้างโฟลเดอร์แนะนำ

```
app/
├── Domain/          # Entities และ Business Rules
├── Application/     # Use Cases, Services
├── Infrastructure/  # Repositories, DB Models, API Clients
└── Http/            # Controllers, Requests, Responses
```

4.2 ตัวอย่างไลฟ์ไซเคิลของคำขอ (Request Lifecycle)

1. **Controller (Interface Layer)** รับคำขอจาก HTTP
2. เรียกใช้ **Service (Application Layer)** เพื่อดำเนินการธุรกิจ
3. Service ใช้ **Entities (Domain Layer)** และ **Repository Interface**
4. Repository Interface ถูก implement ใน **Infrastructure Layer** โดยเชื่อมต่อกับ DB หรือ External APIs
5. ผลลัพธ์ส่งกลับผ่าน Controller ไปยัง Client

4.3 การใช้ Dependency Injection

- ใช้ DI เพื่อ inject Repository Interface เข้าไปใน Service
- Registry Interface กับ Implementation ผ่าน Service Provider
- ทำให้เปลี่ยน implementation ได้ง่ายโดยไม่ต้องแก้ไข Service หรือ Controller

---

5. ตัวอย่างเชิงลึก: User Registration ด้วย Clean Architecture

Domain Layer

```
namespace App\Domain;

class User
{
    private string $name;
    private string $email;

    public function __construct(string $name, string $email)
    {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new \InvalidArgumentException("Invalid email");
        }
        $this->name = $name;
        $this->email = $email;
    }

    // Getter methods...
}
```

Application Layer

```
namespace App\Application;

use App\Domain\User;
use App\Domain\Repositories\UserRepositoryInterface;

class RegisterUserService
{
    private UserRepositoryInterface $userRepository;

    public function __construct(UserRepositoryInterface $repo)
    {
        $this->userRepository = $repo;
    }

    public function register(array $data): User
    {
        $user = new User($data['name'], $data['email']);
        return $this->userRepository->save($user);
    }
}
```

Infrastructure Layer

```
namespace App\Infrastructure\Repositories;

use App\Domain\User;
use App\Domain\Repositories\UserRepositoryInterface;
use App\Models\User as EloquentUser;

class UserRepository implements UserRepositoryInterface
{
    public function save(User $user): User
    {
        $model = new EloquentUser();
        $model->name = $user->getName();
        $model->email = $user->getEmail();
```

```
        $model->save();

        return $user;
    }
}
```

Http Layer (Controller)

```
namespace App\Http\Controllers;

use App\Application\RegisterUserService;
use Illuminate\Http\Request;

class UserController extends Controller
{
    private RegisterUserService $registerService;

    public function __construct(RegisterUserService $service)
    {
        $this->registerService = $service;
    }

    public function register(Request $request)
    {
        $user = $this->registerService->register($request->only('name', 'email'));
        return response()->json($user);
    }
}
```

---

6. ข้อดีของ Clean Architecture ใน Laravel

- ลดการพึ่งพิง Laravel Framework ตรง ๆ ใน Business Logic
- เปลี่ยนแปลงโครงสร้างภายนอกโดยไม่กระทบระบบธุรกิจ
- เพิ่มความชัดเจนในโครงสร้างโปรเจกต์สำหรับทีมขนาดใหญ่
- รองรับ Unit Testing ง่ายเพราะแยก Domain Logic

---

7. สรุป

เรื่อง	สาระสำคัญ
Design Patterns	Singleton, Factory, Strategy ช่วยแก้ปัญหาแบบซ้ำ ๆ และเพิ่มความยืดหยุ่น
Clean Architecture	แยกเลเยอร์เพื่อความชัดเจนและยืดหยุ่นของระบบ
Layers	Domain, Application, Interface, Infrastructure
Laravel	ใช้ Service Container และ DI เพื่อจัดการ dependencies ระหว่างเลเยอร์
ผลลัพธ์	โค้ดมีความ maintainable, scalable, testable

การประยุกต์ใช้ Design Patterns ใน Laravel

1. Singleton Pattern

แนวคิดหลัก

- มี instance เดียวของคลาสในระบบ
- ป้องกันการสร้าง instance ใหม่ซ้ำซ้อน
- ใน Laravel service container จะทำ singleton service ได้ง่ายด้วย singleton() method

ตัวอย่างใน Laravel

ตัวอย่างสร้าง Logger แบบ Singleton

```
<?php
namespace App\Services;

class Logger
{
    private static $instance = null;

    // ป้องกันการสร้าง instance ใหม่จากภายนอก
    private function __construct() {}

    // รับ instance เดียวกันเสมอ
    public static function getInstance()
    {
        if (self::$instance === null) {
            self::$instance = new Logger();
        }
```

```
        return self::$instance;
    }

    public function log($message)
    {
        // บันทึกข้อความลงไฟล์หรือแสดงผล
        echo "Log: " . $message . PHP_EOL;
    }
}
```

ใช้งาน

```
$logger = Logger::getInstance();
$logger->log('User logged in');
```

Laravel Service Container แบบ Singleton

ใน AppServiceProvider สามารถลงทะเบียน service แบบ singleton

```
public function register()
{
    $this->app->singleton('logger', function ($app) {
        return new \App\Services\Logger();
    });
}
```

และเรียกใช้ใน Controller หรือที่อื่น ๆ ผ่าน DI หรือ app('logger')

---

2. Factory Pattern

แนวคิดหลัก

- สร้าง object โดยไม่ต้องรู้รายละเอียดของ class ที่จะ instantiate
- ใช้ factory class หรือ method สร้าง object ตามเงื่อนไข

ตัวอย่างใน Laravel

สร้าง Notification Factory

```
<?php
namespace App\Factories;

interface NotificationInterface
{
    public function send(string $message);
```

```
}

class EmailNotification implements NotificationInterface
{
    public function send(string $message)
    {
        // ส่งอีเมล
        echo "Send Email: $message" . PHP_EOL;
    }
}

class SmsNotification implements NotificationInterface
{
    public function send(string $message)
    {
        // ส่ง SMS
        echo "Send SMS: $message" . PHP_EOL;
    }
}

class NotificationFactory
{
    public static function create(string $type): NotificationInterface
    {
        switch ($type) {
            case 'email':
                return new EmailNotification();
            case 'sms':
                return new SmsNotification();
            default:
                throw new \Exception("Notification type [$type] not supported");
        }
    }
}
```

ใช้งาน

```
$notification = NotificationFactory::create('email');
$notification->send("Hello from Factory!");
```

---

3. Strategy Pattern

แนวคิดหลัก

- กำหนดพฤติกรรมหรือ algorithm ที่เปลี่ยนได้ โดย encapsulate ใน class แยกต่างหาก
- ตัว context จะใช้ strategy ที่ถูกกำหนดใน runtime

ตัวอย่างใน Laravel

สร้าง Strategy Interface และ Implementations

```
<?php
namespace App\Strategies;

interface PaymentStrategy
{
    public function pay(float $amount);
}

class CreditCardPayment implements PaymentStrategy
{
    public function pay(float $amount)
    {
        echo "Paid $amount using Credit Card." . PHP_EOL;
    }
}

class PaypalPayment implements PaymentStrategy
{
    public function pay(float $amount)
    {
        echo "Paid $amount using Paypal." . PHP_EOL;
    }
}
```

Context Class

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

use App\Strategies\PaymentStrategy;

class PaymentContext
{
    private PaymentStrategy $strategy;

    public function __construct(PaymentStrategy $strategy)
    {
        $this->strategy = $strategy;
    }

    public function pay(float $amount)
    {
        $this->strategy->pay($amount);
    }
}
```

ใช้งาน

```php
use App\Context\PaymentContext;
use App\Strategies\PaypalPayment;
use App\Strategies\CreditCardPayment;

$payment = new PaymentContext(new PaypalPayment());
$payment->pay(1500);

$payment = new PaymentContext(new CreditCardPayment());
$payment->pay(3000);
```

สรุป

Pattern	จุดเด่น	ตัวอย่างใน Laravel
Singleton	มี instance เดียวในระบบ	Logger, Cache service, Config service

Pattern	จุดเด่น	ตัวอย่างใน Laravel
Factory	สร้าง object แบบ dynamic ตามชนิด	Notification Factory, Payment Factory
Strategy	เลือกพฤติกรรม/algorithm แบบ runtime	Payment methods, Shipping methods

---

ต่อไปนี้คือ **ตัวอย่างโปรแกรมแบบเต็มไฟล์** พร้อมโครงสร้าง โค้ด และคำอธิบาย รวมทั้งผลการรัน สำหรับ **Design Patterns** 3 แบบ (Singleton, Factory, Strategy) แบ่งเป็น

- 3 โปรแกรมพื้นฐาน (Basic Examples)
- 3 โปรแกรมแนวประยุกต์ (Applied Examples)

ตัวอย่างโปรแกรมพื้นฐาน (Basic Examples)

ตัวอย่าง 1: Singleton Logger Service

โครงสร้างไฟล์

```
app/
 └── Services/
      └── Logger.php
routes/
 └── web.php
```

Logger.php

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

class Logger
{
    private static $instance = null;

    private function __construct() {}

    public static function getInstance()
    {
        if (self::$instance === null) {
            self::$instance = new Logger();
        }
```

```
        return self::$instance;
    }

    public function log($message)
    {
        // สำหรับทดสอบแค่ echo ข้อความ
        echo "Log entry: " . $message;
    }
}
```

web.php (Route)

```
use Illuminate\Support\Facades\Route;
use App\Services\Logger;

Route::get('/singleton-logger', function () {
    $logger = Logger::getInstance();
    $logger->log("User accessed singleton logger.");
});
```

คำอธิบายโค้ด

- คลาส Logger เป็น Singleton ที่สร้าง instance เดียวกันเสมอ
- route /singleton-logger เรียกใช้ Logger instance เดียวกันและแสดงข้อความ

ผลการรัน

เรียก URL: /singleton-logger

ผลลัพธ์: Log entry: User accessed singleton logger.

ตัวอย่าง 2: Factory Pattern — Notification Factory

โครงสร้างไฟล์

```
app/
  └── Factories/
      └── NotificationFactory.php
routes/
  └── web.php
```

NotificationFactory.php

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

```
interface NotificationInterface
{
    public function send(string $message);
}

class EmailNotification implements NotificationInterface
{
    public function send(string $message)
    {
        echo "Send Email: $message";
    }
}

class SmsNotification implements NotificationInterface
{
    public function send(string $message)
    {
        echo "Send SMS: $message";
    }
}

class NotificationFactory
{
    public static function create(string $type): NotificationInterface
    {
        if ($type === 'email') {
            return new EmailNotification();
        } elseif ($type === 'sms') {
            return new SmsNotification();
        } else {
            throw new \Exception("Invalid notification type");
        }
    }
```

```
}
```

web.php (Route)

```
use Illuminate\Support\Facades\Route;
use App\Factories\NotificationFactory;

Route::get('/factory-notification/{type}', function ($type) {
    try {
        $notification = NotificationFactory::create($type);
        $notification->send("This is a factory notification");
    } catch (\Exception $e) {
        return $e->getMessage();
    }
});
```

คำอธิบายโค้ด

- NotificationFactory สร้างอ็อบเจ็กต์ Notification ตามชนิดที่ระบุ
- route /factory-notification/{type} รับ parameter เพื่อเลือกชนิดการแจ้งเตือน

ผลการรัน

- เรียก /factory-notification/email
  ผลลัพธ์: Send Email: This is a factory notification
- เรียก /factory-notification/sms
  ผลลัพธ์: Send SMS: This is a factory notification

ตัวอย่าง 3: Strategy Pattern — Payment Methods

โครงสร้างไฟล์

```
app/
  └── Strategies/
      ├── PaymentStrategy.php
      ├── PaypalPayment.php
      └── CreditCardPayment.php
routes/
  └── web.php
```

PaymentStrategy.php

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

```
interface PaymentStrategy
{
    public function pay(float $amount);
}
```

PaypalPayment.php

```
<?php
namespace App\Strategies;

class PaypalPayment implements PaymentStrategy
{
    public function pay(float $amount)
    {
        echo "Paid $amount using Paypal.";
    }
}
```

CreditCardPayment.php

```
<?php
namespace App\Strategies;

class CreditCardPayment implements PaymentStrategy
{
    public function pay(float $amount)
    {
        echo "Paid $amount using Credit Card.";
    }
}
```

web.php (Route)

```
use Illuminate\Support\Facades\Route;
use App\Strategies\PaypalPayment;
use App\Strategies\CreditCardPayment;

Route::get('/strategy-payment/{method}/{amount}', function ($method, $amount) {
    if ($method === 'paypal') {
```

```
        $payment = new PaypalPayment();
    } elseif ($method === 'creditcard') {
        $payment = new CreditCardPayment();
    } else {
        return "Payment method not supported";
    }
    $payment->pay((float)$amount);
});
```

คำอธิบายโค้ด

- Interface PaymentStrategy กำหนด method pay()
- สองคลาส implements strategy: PaypalPayment, CreditCardPayment
- route /strategy-payment/{method}/{amount} เลือก strategy และจ่ายเงินตามจำนวน

ผลการรัน

- /strategy-payment/paypal/1000
  ผลลัพธ์: Paid 1000 using Paypal.
- /strategy-payment/creditcard/500
  ผลลัพธ์: Paid 500 using Credit Card.

ตัวอย่างโปรแกรมแนวประยุกต์ (Applied Examples)

ตัวอย่าง 4: Singleton + Laravel Service Container — Configuration Service

โครงสร้างไฟล์

```
app/
  └── Services/
      └── ConfigService.php
app/Providers/
  └── AppServiceProvider.php
routes/
  └── web.php
```

ConfigService.php

```
<?php
namespace App\Services;

class ConfigService
```

```
{
    private $config;

    public function __construct()
    {
        // โหลด config จากไฟล์หรือ DB
        $this->config = [
            'app_name' => 'My Laravel App',
            'version' => '1.0.0',
        ];
    }

    public function get($key)
    {
        return $this->config[$key] ?? null;
    }
}
```

AppServiceProvider.php (register singleton)

```
public function register()
{
    $this->app->singleton(ConfigService::class, function ($app) {
        return new ConfigService();
    });
}
```

web.php (Route)

```
use App\Services\ConfigService;
use Illuminate\Support\Facades\Route;

Route::get('/app-config', function (ConfigService $configService) {
    return [
        'app_name' => $configService->get('app_name'),
        'version' => $configService->get('version'),
    ];
});
```

คำอธิบาย

- ConfigService เป็น singleton ผ่าน service container
- สามารถ inject ใช้ใน Controller หรือ Route Closure ได้
- ลดการสร้าง instance ซ้ำ ๆ

ผลการรัน

เรียก /app-config ได้ JSON:

```
{
    "app_name": "My Laravel App",
    "version": "1.0.0"
}
```

ตัวอย่าง 5: Factory Pattern — Payment Gateway Factory (with Laravel Bindings)

โครงสร้างไฟล์

```
app/
  └── Factories/
      └── PaymentGatewayFactory.php
  └── Services/
      ├── PaymentGatewayInterface.php
      ├── PaypalGateway.php
      └── StripeGateway.php
app/Providers/
  └── AppServiceProvider.php
routes/
  └── web.php
```

PaymentGatewayInterface.php

```
<?php
namespace App\Services;

interface PaymentGatewayInterface
{
    public function charge(float $amount);
}
```

PaypalGateway.php

```
<?php
namespace App\Services;

class PaypalGateway implements PaymentGatewayInterface
{
    public function charge(float $amount)
    {
        return "Charging $amount via Paypal";
    }
}
```

StripeGateway.php

```
<?php
namespace App\Services;

class StripeGateway implements PaymentGatewayInterface
{
    public function charge(float $amount)
    {
        return "Charging $amount via Stripe";
    }
}
```

PaymentGatewayFactory.php

```
<?php
namespace App\Factories;

use App\Services\PaypalGateway;
use App\Services\StripeGateway;
use App\Services\PaymentGatewayInterface;

class PaymentGatewayFactory
{
    public static function create(string $type): PaymentGatewayInterface
    {
        switch ($type) {
```

```
            case 'paypal':
                return new PaypalGateway();
            case 'stripe':
                return new StripeGateway();
            default:
                throw new \Exception("Payment gateway [$type] not supported");
        }
    }
}
```

AppServiceProvider.php (optional bindings)

```
public function register()
{
    // สามารถ bind interface กับ implementation ได้ถ้าต้องการ
}
```

web.php (Route)

```
use Illuminate\Support\Facades\Route;
use App\Factories\PaymentGatewayFactory;

Route::get('/pay/{gateway}/{amount}', function ($gateway, $amount) {
    try {
        $paymentGateway = PaymentGatewayFactory::create($gateway);
        return $paymentGateway->charge((float)$amount);
    } catch (\Exception $e) {
        return $e->getMessage();
    }
});
```

ผลการรัน

- /pay/paypal/2500 ⟶ Charging 2500 via Paypal
- /pay/stripe/1500 ⟶ Charging 1500 via Stripe

ตัวอย่าง 6: Strategy Pattern — Dynamic Discount Strategy

โครงสร้างไฟล์

```
app/
└── Strategies/
```

```
    ├── DiscountStrategy.php
    ├── NoDiscount.php
    ├── PercentageDiscount.php
    └── FixedAmountDiscount.php
routes/
└── web.php
```

DiscountStrategy.php

```
<?php
namespace App\Strategies;

interface DiscountStrategy
{
    public function apply(float $amount): float;
}
```

NoDiscount.php

```
<?php
namespace App\Strategies;
class NoDiscount implements DiscountStrategy
{
public function apply(float $amount): float
{
return $amount;
}
}
```

PercentageDiscount.php

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

class PercentageDiscount implements DiscountStrategy
{
    protected $percent;
```