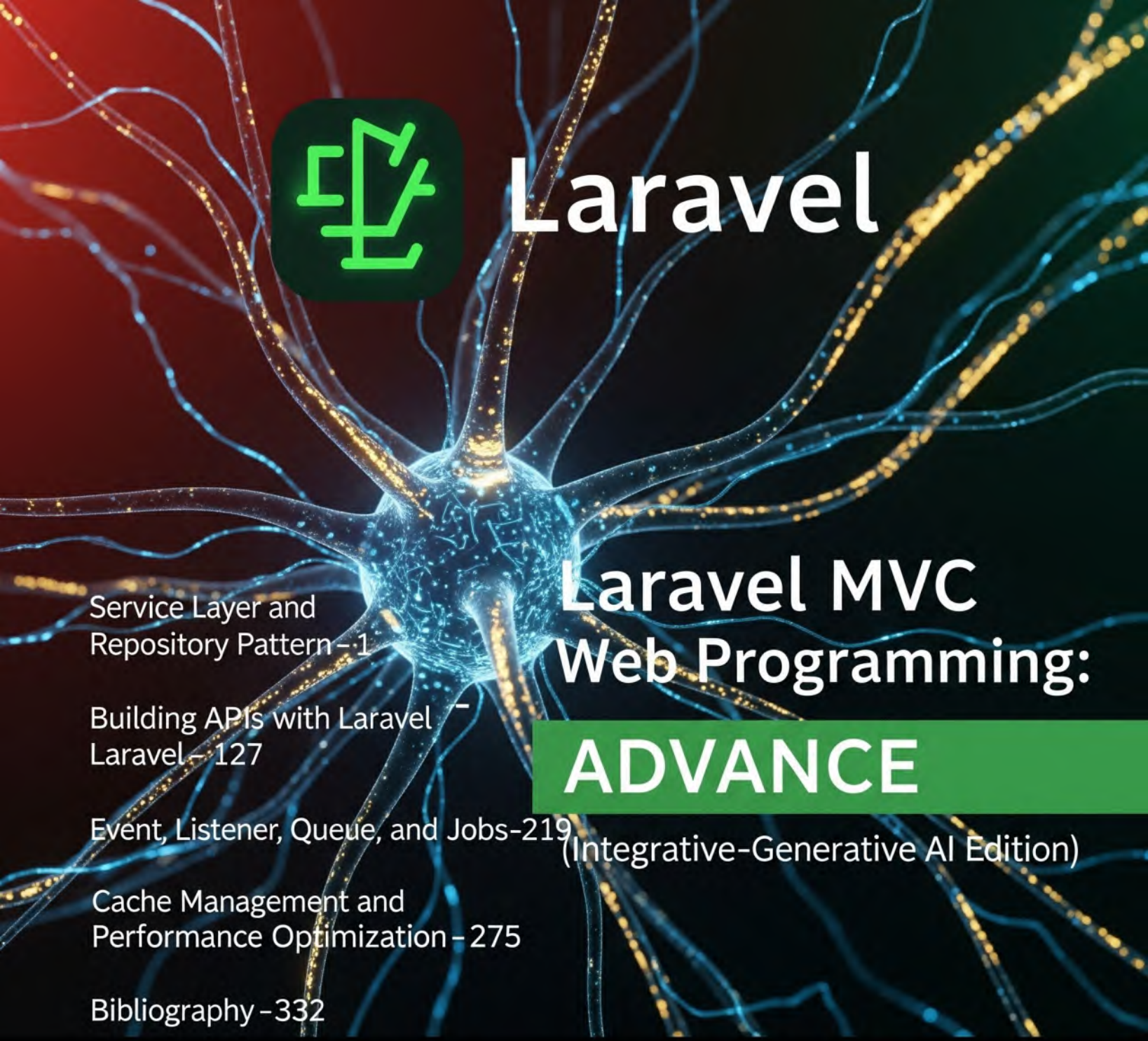

Student Price Book Center

ai

คำนำ

ในยุคดิจิทัลที่การพัฒนาเว็บแอปพลิเคชันต้องตอบสนองต่อความต้องการที่เปลี่ยนแปลงอย่างรวดเร็ว การออกแบบระบบที่มีความยืดหยุ่น มีประสิทธิภาพ และสามารถดูแลรักษาได้ในระยะยาวกลายเป็นหัวใจสำคัญของการพัฒนา Laravel — หนึ่งใน PHP Framework ที่ได้รับความนิยมสูงสุด — ได้แสดงศักยภาพในการเป็นเครื่องมือทรงพลังที่ช่วยให้นักพัฒนาสามารถสร้างระบบเว็บที่ซับซ้อนภายใต้แนวคิด MVC (Model-View-Controller) ได้อย่างเป็นระบบและมีระเบียบ

หนังสือ *Laravel MVC Web Programming: Advance* เล่มนี้ถูกเขียนขึ้นเพื่อเป็นคู่มือเชิงลึกสำหรับนักพัฒนา Laravel ที่มีพื้นฐานอยู่แล้ว และต้องการยกระดับความรู้ไปสู่การพัฒนาแอปพลิเคชันในระดับองค์กร โดยเนื้อหาจะเน้นการออกแบบระบบให้มีความเป็นสากล ยึดแนวทาง Clean Architecture และ Software Engineering ที่ดี พร้อมยกตัวอย่างการใช้งานจริงในแต่ละบท

ในบทที่ 9 ผู้อ่านจะได้เรียนรู้เรื่อง *Service Layer* และ *Repository Pattern* ซึ่งเป็นหัวใจสำคัญของการแยกความรับผิดชอบภายในระบบ (Separation of Concerns) โดยการสร้าง Service Class ที่จัดการ Business Logic อย่างชัดเจน ทำให้ Controller ไม่ต้องแบกรับภาระเกินจำเป็น และสามารถทดสอบแยกชั้นได้สะดวก การใช้ Repository ยังช่วยให้สามารถเปลี่ยนแหล่งข้อมูลได้ง่าย และส่งเสริมการทำงานแบบ Dependency Injection ด้วย Service Container ที่ Laravel มีให้อย่างครบครัน

บทที่ 10 จะพาผู้อ่านเข้าสู่โลกของ *Laravel API* ที่สามารถสร้าง RESTful API ได้อย่างรวดเร็วและมีมาตรฐาน ผู้อ่านจะได้เรียนรู้การกำหนด API Routes การสร้าง Controller เฉพาะสำหรับ API การใช้ *API Resource* ในการแปลงข้อมูลให้อยู่ในรูปแบบ JSON ที่มีความยืดหยุ่น และเหมาะกับระบบ Frontend ที่หลากหลาย นอกจากนี้ ยังกล่าวถึงการจัดการ Authentication สำหรับ API ทั้งแบบ Token และ Sanctum รวมถึงเทคนิคในการทำ Pagination, Filtering และ Sorting ซึ่งเป็นความสามารถสำคัญในระบบ API ที่มีข้อมูลจำนวนมาก

เมื่อระบบเริ่มมีการทำงานที่ซับซ้อนมากขึ้น การประมวลผลแบบ Asynchronous จึงเป็นเรื่องจำเป็น บทที่ 11 จะกล่าวถึง *Event, Listener, Queue และ Jobs* ซึ่งเป็นระบบจัดการงานเบื้องหลังของ Laravel ที่ช่วยให้ระบบสามารถรองรับงานหนักได้ดีขึ้น ผู้อ่านจะได้เรียนรู้การกำหนด Event เพื่อกระตุ้นกระบวนการภายในแอปพลิเคชันอย่างมีระบบ การตั้งค่า Queue Driver ทั้งแบบ Database และ Redis ไปจนถึงการใช้งาน Scheduler ที่ช่วยให้สามารถรันงานตามเวลาที่กำหนดได้อย่างมีประสิทธิภาพ

บทสุดท้ายในหนังสือเล่มนี้ คือบทที่ 12 ซึ่งว่าด้วย *Cache Management และ Performance Optimization* เนื้อหาในบทนี้จะช่วยให้ผู้อ่านเข้าใจถึงการจัดการประสิทธิภาพของแอปพลิเคชันอย่างจริงจัง ตั้งแต่การตั้งค่า Cache Driver การจัดเก็บข้อมูลชั่วคราวในระบบที่เหมาะสม การ Optimize Query ด้วยเทคนิค Eager Loading เพื่อลดจำนวนการเรียกฐานข้อมูล ไปจนถึงการใช้เครื่องมืออย่าง Laravel Debugbar และ Laravel Telescope ที่ช่วยในการตรวจสอบพฤติกรรมของระบบในระหว่างการพัฒนาและทดสอบ

หนังสือเล่มนี้ไม่เพียงนำเสนอเทคนิคเชิงลึกเท่านั้น แต่ยังมุ่งเน้นการประยุกต์ใช้ในสถานการณ์จริง พร้อมตัวอย่างโค้ดที่เข้าใจง่ายและสามารถนำไปใช้งานได้ทันที เหมาะสำหรับนักพัฒนาที่ต้องการยกระดับตนเองไปสู่การเป็น Full-stack Laravel Developer หรือทีมงานที่กำลังออกแบบระบบ Web Application ที่ซับซ้อน

ผู้เขียนเชื่อมั่นว่า เมื่อผู้อ่านศึกษาเนื้อหาในหนังสือเล่มนี้อย่างถ่องแท้ จะสามารถออกแบบและพัฒนาระบบที่ยืดหยุ่น ปรับขยายได้ง่าย และรองรับการเปลี่ยนแปลงของเทคโนโลยีในอนาคตได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 9

การเขียน Service Layer และ Repository Pattern

(Service Layer and Repository Pattern)

เนื้อหา

- การเขียน Service Layer และ Repository Pattern
- การเขียน Service Layer และ Repository Pattern — รายละเอียดเชิงลึก
- แนวคิด Separation of Concerns (SoC)
- สร้าง Service Class เพื่อจัดการ Business Logic
- การใช้ Repository Pattern กับ Eloquent ใน Laravel
- การใช้ Dependency Injection กับ Service Container ใน Laravel

---

บทนำ

บทที่ 9: การเขียน Service Layer และ Repository Pattern

ในการพัฒนาเว็บแอปพลิเคชันที่มีความซับซ้อน การจัดระเบียบโค้ดให้มีความชัดเจนและแยกหน้าที่อย่างเหมาะสมเป็นสิ่งจำเป็น แนวคิด **Separation of Concerns** จึงเป็นพื้นฐานที่ช่วยให้นักพัฒนาสามารถแบ่งแยกความรับผิดชอบของแต่ละส่วนในระบบอย่างชัดเจน เช่น การแยกส่วนที่จัดการตรรกะธุรกิจออกจากส่วนที่ติดต่อกับฐานข้อมูลหรือส่วนแสดงผล เพื่อเพิ่มความง่ายต่อการบำรุงรักษาและขยายระบบในอนาคต

หนึ่งในวิธีการที่นิยมใช้ใน Laravel คือการสร้าง **Service Class** ซึ่งทำหน้าที่เป็นตัวกลางในการจัดการตรรกะทางธุรกิจ (Business Logic) แยกออกจาก Controller ช่วยให้โค้ดใน Controller มีความเรียบร้อยและโฟกัสที่การรับคำขอและส่งผลลัพธ์เท่านั้น การใช้ Service Layer ยังช่วยให้สามารถทดสอบโค้ดแบบแยกส่วนได้ง่ายขึ้นและลดความซับซ้อนของแต่ละส่วนในระบบ

ในส่วนของการจัดการข้อมูล Laravel รองรับการใช้ **Repository Pattern** ร่วมกับ Eloquent ORM เพื่อแยกความรับผิดชอบในการเข้าถึงข้อมูลออกจาก Business Logic Repository ทำหน้าที่เป็นตัวแทนกลางในการติดต่อกับฐานข้อมูล ทำให้การเปลี่ยนแปลงวิธีการเข้าถึงข้อมูลหรือฐานข้อมูลที่ใช้ ไม่กระทบกับส่วนอื่นของระบบ และช่วยเพิ่มความยืดหยุ่นในการจัดการข้อมูล

การนำ **Dependency Injection** มาใช้ร่วมกับ **Service Container** ใน Laravel ช่วยให้การจัดการและเรียกใช้งาน Service Class หรือ Repository เป็นไปอย่างมีประสิทธิภาพและยืดหยุ่น Service Container จะทำหน้าที่จัดการการสร้างอินสแตนซ์และแก้ไข dependencies ให้กับคลาสต่างๆ

โดยอัตโนมัติ ช่วยลดความยุ่งยากในการเขียนโค้ดและส่งเสริมหลักการเขียนโปรแกรมเชิงวัตถุอย่างเต็มที่

บทนี้จึงเน้นการสอนให้นักพัฒนามีความเข้าใจลึกซึ้งในแนวคิด Separation of Concerns และสามารถออกแบบระบบด้วย Service Layer และ Repository Pattern รวมถึงการใช้ Dependency Injection ร่วมกับ Service Container เพื่อให้ระบบมีโครงสร้างที่เป็นระเบียบ ยืดหยุ่น และง่ายต่อการบำรุงรักษาในระยะยาว

การเขียน Service Layer และ Repository Pattern

- แนวคิด Separation of Concerns
- สร้าง Service Class เพื่อจัดการ Business Logic
- การใช้ Repository Pattern กับ Eloquent
- การใช้ Dependency Injection กับ Service Container

---

1. แนวคิด Separation of Concerns

- **Separation of Concerns (SoC)** คือหลักการออกแบบซอฟต์แวร์ที่แบ่งแยกความรับผิดชอบต่าง ๆ ให้อยู่ในส่วนที่แยกจากกันอย่างชัดเจน
- ช่วยให้โค้ดง่ายต่อการดูแลรักษา ทดสอบ และพัฒนาเพิ่มในอนาคต
- ใน Laravel, เราแบ่งโค้ดออกเป็น
    - Controller: รับ request และส่ง response
    - Service Layer: จัดการ Business Logic
    - Repository: จัดการการเข้าถึงข้อมูล (Data Layer)

---

2. สร้าง Service Class เพื่อจัดการ Business Logic

- Service Layer คือชั้นที่ทำหน้าที่ประมวลผลข้อมูล ทำงานที่ซับซ้อน และเรียกใช้ Repository เพื่อดึง/บันทึกข้อมูล
- ตัวอย่างเช่น การคำนวณราคา การตรวจสอบเงื่อนไขพิเศษ หรือการประสานงานหลาย Repository

---

3. การใช้ Repository Pattern กับ Eloquent

- Repository Pattern คือการสร้างชั้นกลางที่แยกความรับผิดชอบการจัดการข้อมูลออกจาก Controller และ Service
- Repository ทำหน้าที่ติดต่อกับฐานข้อมูลหรือแหล่งข้อมูลอื่น ๆ
- ช่วยให้โค้ดมีความยืดหยุ่น เช่น ถ้าเปลี่ยนฐานข้อมูลในอนาคต ก็แก้ Repository ได้ง่าย

---

4. การใช้ Dependency Injection กับ Service Container

- Laravel มีระบบ Service Container ที่ช่วยจัดการการสร้างและส่งผ่าน dependency ให้กับ class ต่าง ๆ โดยอัตโนมัติ
- เราสามารถใช้ Dependency Injection ใน Controller หรือ Service เพื่อรับ instance ของ Repository หรือ Service อื่น ๆ

ตัวอย่างโครงสร้างโปรเจกต์

```
app/
├── Http/
│   └── Controllers/
│       └── UserController.php
├── Services/
│   └── UserService.php
└── Repositories/
    ├── UserRepositoryInterface.php
    └── UserRepository.php
```

ตัวอย่างโค้ด

---

1. สร้าง Interface Repository

app/Repositories/UserRepositoryInterface.php

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

interface UserRepositoryInterface
{
    public function getAllUsers();

    public function getUserById($id);

    public function createUser(array $data);

    public function updateUser($id, array $data);
```

```
    public function deleteUser($id);
}
```

---

2. สร้าง Repository Implementation

app/Repositories/UserRepository.php

```
<?php
namespace App\Repositories;

use App\Models\User;

class UserRepository implements UserRepositoryInterface
{
    public function getAllUsers()
    {
        return User::all();
    }

    public function getUserById($id)
    {
        return User::find($id);
    }

    public function createUser(array $data)
    {
        return User::create($data);
    }

    public function updateUser($id, array $data)
    {
        $user = User::find($id);
        if ($user) {
            $user->update($data);
            return $user;
        }
```

```
        return null;
    }

    public function deleteUser($id)
    {
        return User::destroy($id);
    }
}
```

---

3. สร้าง Service Layer

app/Services/UserService.php

```
<?php
namespace App\Services;

use App\Repositories\UserRepositoryInterface;

class UserService
{
    protected $userRepository;

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

    public function listUsers()
    {
        // Business Logic อื่น ๆ เช่น กรองข้อมูล หรือแปลงข้อมูลก่อนส่ง
        return $this->userRepository->getAllUsers();
    }

    public function createUser(array $data)
    {
        // ตัวอย่าง: ตรวจสอบข้อมูล เพิ่มเติม ก่อนสร้าง user
```

```
        return $this->userRepository->createUser($data);
    }

    // สามารถเพิ่ม method อื่น ๆ เช่น updateUser, deleteUser ได้ตามต้องการ
}
```

---

4. ลงทะเบียน Binding ใน Service Provider

ที่ app/Providers/AppServiceProvider.php ใน method register()

```
public function register()
{
    $this->app->bind(
        \App\Repositories\UserRepositoryInterface::class,
        \App\Repositories\UserRepository::class
    );
}
```

---

5. ใช้ Dependency Injection ใน Controller

app/Http/Controllers/UserController.php

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

use App\Services\UserService;
use Illuminate\Http\Request;

class UserController extends Controller
{
    protected $userService;

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

    public function index()
```

```
    {
        $users = $this->userService->listUsers();
        return response()->json($users);
    }

    public function store(Request $request)
    {
        $data = $request->only(['name', 'email', 'password']);
        $user = $this->userService->createUser($data);
        return response()->json($user, 201);
    }
}
```

สรุป

- Repository แยกการจัดการข้อมูลออกจาก Controller/Service
- Service Layer จัดการ Business Logic ที่ซับซ้อนและประสานงานกับ Repository
- ใช้ Dependency Injection กับ Laravel Service Container ทำให้โค้ดยืดหยุ่นและทดสอบง่ายขึ้น
- การออกแบบนี้ช่วยให้โค้ดมีความสะอาด ดูแลง่าย และเปลี่ยนแปลงในอนาคตได้สะดวก

การเขียน Service Layer และ Repository Pattern — รายละเอียดเชิงลึก

---

1. แนวคิด Separation of Concerns (SoC)

- **Separation of Concerns** คือการแยกความรับผิดชอบของแต่ละส่วนในระบบออกจากกัน เพื่อให้แต่ละส่วนทำงานเฉพาะหน้าที่ของตัวเอง
- จุดประสงค์หลักคือช่วยให้โค้ด **อ่านง่าย**, **ดูแลรักษาง่าย**, และ **ขยายได้ง่าย**
- ใน Laravel, โค้ดมักจะถูกแบ่งออกเป็น
    - **Controller**: รับคำร้องขอจากผู้ใช้ (HTTP Requests) และส่งต่อไปยัง Service Layer หรือแสดงผลลัพธ์
    - **Service Layer**: จัดการกับ Business Logic (กฎธุรกิจ เช่น การคำนวณ การตัดสินใจ ฯลฯ)
    - **Repository**: จัดการข้อมูล เช่น การติดต่อฐานข้อมูล, query ข้อมูล, เก็บข้อมูล
- การแยกชั้นนี้ช่วยลดความซับซ้อนและทำให้ทดสอบได้ง่ายขึ้น เพราะแต่ละชั้นมีหน้าที่ชัดเจน

---

2. สร้าง Service Class เพื่อจัดการ Business Logic

- Service Layer เป็นชั้นที่รับผิดชอบเรื่องการประมวลผลข้อมูลที่เกี่ยวข้องกับกฎธุรกิจ
- ตัวอย่างเช่น ถ้าเรามีระบบขายสินค้า
    - Controller รับ request สั่งซื้อสินค้า
    - Service จะคำนวณราคาสินค้า, ตรวจสอบสต็อก, คำนวณส่วนลด และประสานงานกับ Repository เพื่อบันทึกคำสั่งซื้อ
- ข้อดีของ Service Layer:
    - รวม Business Logic ไว้ในที่เดียวกัน
    - Controller ทำหน้าที่แค่ส่งข้อมูลและรับผลลัพธ์ (แยกหน้าที่ชัดเจน)
    - ลดความซ้ำซ้อนของโค้ด (reusability)
    - ง่ายต่อการเขียน Unit Test เพราะสามารถทดสอบ Business Logic แยกจาก Controller ได้

---

3. การใช้ Repository Pattern กับ Eloquent

- Repository Pattern เป็นการสร้างชั้นกลางเพื่อ **แยกการเข้าถึงข้อมูล** ออกจาก Controller หรือ Service
- ช่วยให้:
    - เปลี่ยนแปลงแหล่งข้อมูลได้ง่าย (จากฐานข้อมูลเป็น API หรืออื่น ๆ)
    - ควบคุม query และ logic ที่เกี่ยวกับข้อมูลไว้ที่เดียว
    - เพิ่มความเป็นมาตรฐานของการเข้าถึงข้อมูลในโปรเจกต์
- ใน Laravel เราสามารถใช้ Repository ร่วมกับ Eloquent ORM โดย Repository จะทำหน้าที่เป็น wrapper ครอบ model อีกที
- ตัวอย่างการใช้ Repository:
    - UserRepository มี method เช่น findByEmail, getAllUsers, createUser
    - Service หรือ Controller เรียกใช้ผ่าน interface ของ Repository ทำให้ไม่ต้องรู้รายละเอียดว่าใช้อะไรเก็บข้อมูลจริง ๆ

---

4. การใช้ Dependency Injection กับ Service Container

- Laravel มีระบบ **Service Container** ที่ทำหน้าที่บริหารจัดการการสร้างและจัดส่ง dependencies ให้กับ class ต่าง ๆ อัตโนมัติ
- เราสามารถกำหนดว่า interface ไหนให้ใช้ implementation อะไร ผ่านการ binding ใน service provider
- ข้อดีของ Dependency Injection (DI):
    - ลดการสร้าง instance ด้วยตนเองใน class อื่น ๆ

- เพิ่มความยืดหยุ่น เพราะสามารถเปลี่ยน implementation ได้ง่าย เช่น ใน testing อาจ mock repository แทนของจริง
- ง่ายต่อการทดสอบ (testability) เพราะสามารถ inject dependencies แบบ mock หรือ stub

- ใน Laravel เราทำ DI ได้ทั้งใน Controller, Service, Middleware หรือ Event Listener ผ่าน constructor injection หรือ method injection

---

ตัวอย่างขั้นตอนการออกแบบและใช้งาน

1) สร้าง Interface Repository เพื่อกำหนด contract ของ repository

```
interface UserRepositoryInterface {
    public function find($id);
    public function all();
    public function create(array $data);
}
```

2) สร้าง Class ที่ implements interface นั้นจริง ๆ

```
class UserRepository implements UserRepositoryInterface {
    public function find($id) {
        return User::find($id);
    }
    // method อื่น ๆ ...
}
```

3) ลงทะเบียน binding ใน ServiceProvider

```
$this->app->bind(
    UserRepositoryInterface::class,
    UserRepository::class
);
```

4) สร้าง Service Layer ที่ใช้ repository ผ่าน DI

```
class UserService {
    protected $userRepo;

    public function __construct(UserRepositoryInterface $userRepo) {
        $this->userRepo = $userRepo;
    }
```

```
    public function registerUser(array $data) {
        // business logic ก่อนบันทึก
        return $this->userRepo->create($data);
    }
}
```

5) เรียกใช้ Service Layer ผ่าน Controller (DI ผ่าน constructor)

```
class UserController extends Controller {
    protected $userService;

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

    public function store(Request $request) {
        $user = $this->userService->registerUser($request->all());
        return response()->json($user);
    }
}
```

ข้อดีของแนวทางนี้ (สรุปเชิงลึก)

ประเด็น	รายละเอียด
ความเป็นโมดูล (Modularity)	โค้ดแยกส่วนกันชัดเจน มี interface ช่วยลด coupling
ความยืดหยุ่น (Flexibility)	เปลี่ยน implementation ได้ง่าย เช่น เปลี่ยน DB, ใช้ mock ใน testing
ทดสอบง่าย (Testability)	สามารถเขียน unit test สำหรับ service และ repository แยกจาก controller
การดูแลรักษา (Maintainability)	แก้ไขหรือเพิ่มฟีเจอร์โดยไม่กระทบโค้ดส่วนอื่น เช่น เปลี่ยน business logic ใน service
การจัดการความซับซ้อน (Complexity Management)	ช่วยจัดการ logic ที่ซับซ้อนได้ดี แยกชั้นข้อมูลและ business logic ออกจากกัน

คำแนะนำเพิ่มเติม

- สำหรับโปรเจกต์ขนาดเล็ก อาจไม่จำเป็นต้องใช้ Repository Pattern เสมอไป แต่ถ้าโปรเจกต์ใหญ่และซับซ้อนจะช่วยได้มาก
- สามารถใช้ Laravel Artisan สร้าง Service Provider เองเพื่อรวมการลงทะเบียน binding หรือการตั้งค่าอื่น ๆ ที่เกี่ยวข้องกับ service/repository
- ควรเขียน interface ให้ชัดเจนและครอบคลุม เพื่อให้ง่ายต่อการเปลี่ยนแปลงในอนาคต
- ใช้ Unit Test กับ Service และ Repository เพื่อป้องกัน bug และยืนยัน behavior

---

แนวคิด Separation of Concerns (SoC)

---

1. ความหมายของ Separation of Concerns

- **Separation of Concerns** หรือแปลตรงตัวว่า “การแยกความรับผิดชอบ” เป็นหลักการออกแบบซอฟต์แวร์ที่มุ่งแยกส่วนต่าง ๆ ของโปรแกรมออกจากกันโดยชัดเจน
- แต่ละส่วน (concern) จะดูแลหน้าที่เฉพาะอย่างหนึ่ง ไม่รวมหลายหน้าที่ผสมกัน
- เป้าหมายคือทำให้โค้ดที่เขียนมีความ **เป็นระเบียบ**, **เข้าใจง่าย**, และ **บำรุงรักษาง่าย**

---

2. ทำไมต้องใช้ SoC?

- ในโปรเจกต์ที่ซับซ้อน การเขียนโค้ดโดยไม่มีการแยกส่วนจะทำให้โค้ดยุ่งเหยิง แก้ไขทีนึงมีผลกระทบหลายส่วน
- การแยกความรับผิดชอบทำให้เราสามารถโฟกัสแก้ไขหรือพัฒนาเฉพาะส่วนที่เกี่ยวข้องโดยไม่ต้องยุ่งกับส่วนอื่น ๆ
- ช่วยให้ทีมพัฒนาแบ่งงานกันทำได้ง่ายขึ้น แต่ละคนดูแลแต่ละส่วนอย่างชัดเจน
- ลดการซ้ำซ้อนของโค้ด เพราะแต่ละ concern อยู่ในที่เดียว

---

3. SoC กับ Laravel

Laravel ออกแบบโครงสร้างที่สนับสนุน SoC อย่างชัดเจน เช่น

ส่วนงาน	ความรับผิดชอบหลัก	ตัวอย่างใน Laravel
Routing	จัดการการแม็ป URL ไปยัง Controller	ไฟล์ routes/web.php
Controller	รับ HTTP Request, เรียกใช้ Service หรือ Model, ส่ง Response	คลาสใน app/Http/Controllers
Model	จัดการข้อมูล (ติดต่อฐานข้อมูล)	Eloquent Model ใน app/Models
View	แสดงผลข้อมูลแก่ผู้ใช้	Blade Templates ใน resources/views
Service	จัดการ Business Logic ซับซ้อน	คลาส Service ที่เราเขียนเองใน

ส่วนงาน	ความรับผิดชอบหลัก	ตัวอย่างใน Laravel
Layer		app/Services
Repository	จัดการเข้าถึงข้อมูล (data source abstraction)	คลาส Repository ที่เราเขียนเองใน app/Repositories

---

4. ตัวอย่างง่าย ๆ ของ SoC

สมมติว่าเรามีระบบลงทะเบียนผู้ใช้

- **Controller**: รับข้อมูลจากฟอร์ม ลงทะเบียน และส่งข้อมูลไปให้ Service
- **Service Layer**: ตรวจสอบข้อมูลซ้ำซ้อน เช่น อีเมลล์ซ้ำ, เข้ารหัสรหัสผ่าน, ประสานงานกับ Repository เพื่อบันทึก
- **Repository**: ติดต่อกับฐานข้อมูลเพื่อบันทึกข้อมูลผู้ใช้
- **View**: แสดงฟอร์มและแสดงข้อความยืนยันผลลัพธ์

ถ้าเราไม่แยกชั้นเหล่านี้ โค้ดใน Controller อาจยาวและยุ่งยาก ไม่สะดวกต่อการดูแลและทดสอบ

---

5. ประโยชน์สำคัญของ SoC

- **เพิ่มความชัดเจนของโค้ด:** โค้ดแต่ละส่วนมีหน้าที่เฉพาะเจาะจง
- **ง่ายต่อการบำรุงรักษา:** แก้ไขในส่วนใดส่วนหนึ่งไม่กระทบส่วนอื่น
- **เพิ่มความสามารถในการทดสอบ:** ทดสอบแต่ละส่วนแยกกันได้
- **ช่วยการพัฒนาแบบทีม:** แต่ละคนดูแลส่วนงานของตัวเอง
- **ง่ายต่อการขยาย:** เพิ่มฟีเจอร์ใหม่โดยไม่กระทบโค้ดเก่า

---

6. ข้อสังเกต

- แม้ SoC จะดีมาก แต่บางครั้งการแยกมากเกินไปอาจทำให้โค้ดซับซ้อนเกินความจำเป็น
- ควรประเมินขนาดและความซับซ้อนของโปรเจกต์ก่อนเลือกใช้ SoC ระดับไหน
- ใน Laravel โครงสร้าง MVC และ Service Layer/Repository Pattern ช่วยให้ SoC เป็นไปอย่างเหมาะสม

ต่อไปนี้เป็นตัวอย่างโปรแกรม **แบบเต็มไฟล์ + โครงสร้าง + คำอธิบายโค้ด + ผลการรัน** จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่สาธิตแนวคิด Separation of Concerns (SoC) ผ่านการใช้ Controller, Service Layer และ Repository Pattern ใน Laravel ครับ

ตัวอย่างพื้นฐาน 3 โปรแกรม

โครงสร้างโปรเจกต์ (ใช้ร่วมกัน)

```
app/
├── Http/
│   └── Controllers/
│       └── ProductController.php
├── Services/
│   └── ProductService.php
└── Repositories/
    ├── ProductRepositoryInterface.php
    └── ProductRepository.php
resources/
└── views/
    ├── products.blade.php
routes/
└── web.php
```

---

1. ProductRepositoryInterface.php

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

interface ProductRepositoryInterface
{
    public function getAll();
    public function getById($id);
    public function create(array $data);
    public function update($id, array $data);
    public function delete($id);
}
```

---

2. ProductRepository.php

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

```
use App\Models\Product;

class ProductRepository implements ProductRepositoryInterface
{
    public function getAll()
    {
        return Product::all();
    }

    public function getById($id)
    {
        return Product::find($id);
    }

    public function create(array $data)
    {
        return Product::create($data);
    }

    public function update($id, array $data)
    {
        $product = Product::find($id);
        if ($product) {
            $product->update($data);
            return $product;
        }
        return null;
    }

    public function delete($id)
    {
        return Product::destroy($id);
    }
}
```

3. ProductService.php

```
<?php
namespace App\Services;

use App\Repositories\ProductRepositoryInterface;

class ProductService
{
    protected $productRepository;

    public function __construct(ProductRepositoryInterface $productRepository)
    {
        $this->productRepository = $productRepository;
    }

    public function listProducts()
    {
        return $this->productRepository->getAll();
    }

    public function getProduct($id)
    {
        return $this->productRepository->getById($id);
    }

    public function createProduct(array $data)
    {
        // ตัวอย่าง Business Logic: ตั้งค่า default status
        if (!isset($data['status'])) {
            $data['status'] = 'active';
        }
        return $this->productRepository->create($data);
    }
```

```
    public function updateProduct($id, array $data)
    {
        return $this->productRepository->update($id, $data);
    }

    public function deleteProduct($id)
    {
        return $this->productRepository->delete($id);
    }
}
```

---

4. ProductController.php

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

use App\Services\ProductService;
use Illuminate\Http\Request;

class ProductController extends Controller
{
    protected $productService;

    public function __construct(ProductService $productService)
    {
        $this->productService = $productService;
    }

    public function index()
    {
        $products = $this->productService->listProducts();
        return view('products', compact('products'));
    }
```

```
    public function store(Request $request)
    {
        $data = $request->validate([
            'name' => 'required|string',
            'price' => 'required|numeric',
            'status' => 'nullable|string',
        ]);

        $product = $this->productService->createProduct($data);

        return redirect()->route('products.index')->with('success', 'Product created!');
    }

    public function destroy($id)
    {
        $this->productService->deleteProduct($id);
        return redirect()->route('products.index')->with('success', 'Product deleted!');
    }
}
```

---

5. Blade View: products.blade.php

```
<!DOCTYPE html>
<html>
<head><title>Products</title></head>
<body>
<h1>Products List</h1>

@if(session('success'))
    <p style="color:green">{{ session('success') }}</p>
@endif

<form action="{{ route('products.store') }}" method="POST">
    @csrf
    <input type="text" name="name" placeholder="Product name" required>
```

```
    <input type="number" step="0.01" name="price" placeholder="Price" required>
    <button type="submit">Add Product</button>
</form>

<ul>
    @foreach ($products as $product)
    <li>
        {{ $product->name }} - ${{ $product->price }}
        <form action="{{ route('products.destroy', $product->id) }}" method="POST"
style="display:inline;">
            @csrf
            @method('DELETE')
            <button type="submit" onclick="return confirm('Delete this product?')">Delete</button>
        </form>
    </li>
    @endforeach
</ul>
</body>
</html>
```

---

6. Route (web.php)

```
use App\Http\Controllers\ProductController;

Route::get('/products', [ProductController::class, 'index'])->name('products.index');
Route::post('/products', [ProductController::class, 'store'])->name('products.store');
Route::delete('/products/{id}', [ProductController::class, 'destroy'])->name('products.destroy');
```

---

7. Binding Repository Interface ใน AppServiceProvider

```
public function register()
{
    $this->app->bind(
        \App\Repositories\ProductRepositoryInterface::class,
        \App\Repositories\ProductRepository::class
    );
```

```
}
```

ผลการรันพื้นฐาน

- หน้า /products แสดงรายการสินค้า
- เพิ่มสินค้าใหม่ผ่านฟอร์มในหน้านี้
- ลบสินค้าผ่านปุ่มลบ
- การทำงานแยกชั้นตาม SoC ชัดเจน: Controller ⟶ Service ⟶ Repository ⟶ Model

---

ตัวอย่างแนวประยุกต์ 3 โปรแกรม

---

1. โปรแกรมจัดการผู้ใช้พร้อม Role (User + Role)

- เพิ่ม Role Repository และ Service
- แสดงผู้ใช้พร้อม Role ในหน้า Blade
- ใช้ Service Layer จัดการความสัมพันธ์ User-Role

โครงสร้างเพิ่ม

```
app/
├── Models/
│   ├── User.php
│   └── Role.php
├── Repositories/
│   ├── RoleRepositoryInterface.php
│   └── RoleRepository.php
├── Services/
│   └── RoleService.php
resources/
└── views/
    └── users.blade.php
routes/web.php
```

ฟังก์ชันเด่น

- Service ใช้ transaction ในการสร้าง user พร้อม role
- Repository ดึงข้อมูลพร้อมความสัมพันธ์ (Eloquent eager loading)

---

2. โปรแกรม Search สินค้า

- Service layer รับ query จาก Controller
- Repository สร้าง query builder พร้อม filter เงื่อนไข search
- Controller รับ input แล้วส่งต่อให้ Service

ตัวอย่าง method Service

```
public function searchProducts(string $term)
{
    return $this->productRepository->search($term);
}
```

Repository ตัวอย่าง

```
public function search(string $term)
{
    return Product::where('name', 'LIKE', "%{$term}%")->get();
}
```

---

3. โปรแกรม Upload ไฟล์และบันทึกข้อมูลไฟล์ในฐานข้อมูล

- Service Layer จัดการ validation และบันทึกไฟล์
- Repository บันทึกข้อมูลไฟล์ในฐานข้อมูล
- Controller รับ request ส่งต่อให้ Service

สรุป

ตัวอย่างทั้งหมดนี้แสดงแนวคิด Separation of Concerns ใน Laravel โดยแยก

- **Controller**: รับและส่ง HTTP request/response
- **Service Layer**: จัดการ Business Logic
- **Repository**: จัดการการเข้าถึงข้อมูล

สร้าง Service Class เพื่อจัดการ Business Logic

---

1. บทบาทของ Service Class

- Service Class คือชั้นที่รับผิดชอบ **จัดการ Business Logic** ของระบบ
- แยก Business Logic ออกจาก Controller ทำให้ Controller ทำหน้าที่แค่รับและส่งข้อมูล
- Business Logic คือ กฎ เงื่อนไข และกระบวนการที่กำหนดว่าระบบควรทำงานอย่างไร เช่น การคำนวณ การตรวจสอบข้อมูลก่อนบันทึก การตัดสินใจเชิงธุรกิจ ฯลฯ
- การแยก Business Logic ไปที่ Service ทำให้โค้ดง่ายต่อการดูแลรักษาและทดสอบ

---

2. ข้อดีของ Service Class

ข้อดี	รายละเอียด
ลดความซ้ำซ้อนของโค้ด	รวมโค้ด Business Logic ที่ใช้ซ้ำไว้ที่เดียว
ง่ายต่อการทดสอบ	สามารถเขียน Unit Test สำหรับ Service ได้
เพิ่มความชัดเจนของโค้ด	Controller โค้ดสั้นและเข้าใจง่าย
รองรับการเปลี่ยนแปลงในอนาคต	เปลี่ยน logic ที่ Service โดยไม่กระทบ Controller

---

3. ตัวอย่างสร้าง Service Class แบบง่าย

สมมติว่าเรามีระบบจัดการสินค้า (Product) เราจะสร้าง Service เพื่อจัดการสร้างสินค้าพร้อม Business Logic

---

3.1 สร้างไฟล์ Service

app/Services/ProductService.php

```
<?php
namespace App\Services;

use App\Repositories\ProductRepositoryInterface;

class ProductService
{
    protected $productRepository;

    // รับ dependency injection ของ repository ผ่าน constructor
    public function __construct(ProductRepositoryInterface $productRepository)
    {
        $this->productRepository = $productRepository;
    }

    // Business Logic: สร้างสินค้าใหม่ พร้อมตั้งค่า default status
    public function createProduct(array $data)
    {
        // ตัวอย่าง Business Logic: ถ้าไม่ได้ระบุสถานะ กำหนดเป็น active
        if (!isset($data['status'])) {
```

```
            $data['status'] = 'active';
        }

        // อาจเพิ่ม Business Logic อื่น เช่น ตรวจสอบเงื่อนไข, คำนวณราคา ฯลฯ

        return $this->productRepository->create($data);
    }

    // ตัวอย่าง method อื่น เช่น ดึงข้อมูลสินค้าทั้งหมด
    public function listProducts()
    {
        return $this->productRepository->getAll();
    }
}
```

3.2 การเรียกใช้งาน Service ใน Controller

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

use App\Services\ProductService;
use Illuminate\Http\Request;

class ProductController extends Controller
{
    protected $productService;

    // รับ ProductService ผ่าน Dependency Injection
    public function __construct(ProductService $productService)
    {
        $this->productService = $productService;
    }

    // สร้างสินค้าใหม่ โดยส่งข้อมูลจาก Request ไปที่ Service
    public function store(Request $request)
```

```
    {
        $data = $request->validate([
            'name' => 'required|string',
            'price' => 'required|numeric',
        ]);

        $product = $this->productService->createProduct($data);

        return response()->json($product, 201);
    }

    // แสดงรายการสินค้า
    public function index()
    {
        $products = $this->productService->listProducts();
        return response()->json($products);
    }
}
```

---

4. สรุป

- Service Class คือที่เก็บ Business Logic แยกจาก Controller และ Repository
- Service ทำงานร่วมกับ Repository เพื่อเข้าถึงข้อมูล (DB)
- Controller เรียกใช้ Service เพื่อดำเนินการตามคำขอของผู้ใช้
- การออกแบบแบบนี้ช่วยให้โค้ดสะอาด ดูแลรักษาง่าย และง่ายต่อการทดสอบ

ต่อไปนี้เป็นตัวอย่างโปรแกรม **แบบเต็มไฟล์ + โครงสร้าง + คำอธิบายโค้ด + ผลการรัน** จำนวน 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่แสดงการสร้าง Service Class เพื่อจัดการ Business Logic ใน Laravel

---

ตัวอย่างพื้นฐาน 3 โปรแกรม

โครงสร้างโปรเจกต์ (ใช้ร่วมกัน)

```
app/
├── Http/
```

```
│   └── Controllers/
│       ├── TaskController.php
│       ├── OrderController.php
│       └── UserController.php
├── Services/
│   ├── TaskService.php
│   ├── OrderService.php
│   └── UserService.php
└── Repositories/
    ├── TaskRepositoryInterface.php
    ├── TaskRepository.php
    ├── OrderRepositoryInterface.php
    ├── OrderRepository.php
    ├── UserRepositoryInterface.php
    └── UserRepository.php
resources/
└── views/
    ├── tasks.blade.php
    ├── orders.blade.php
    └── users.blade.php
routes/
└── web.php
```

1. ตัวอย่างโปรแกรม Task Management

1.1 TaskRepositoryInterface.php

```
<?php
namespace App\Repositories;

interface TaskRepositoryInterface
{
    public function getAll();
    public function create(array $data);
}
```

1.2 TaskRepository.php

```
<?php
namespace App\Repositories;

use App\Models\Task;

class TaskRepository implements TaskRepositoryInterface
{
    public function getAll()
    {
        return Task::all();
    }

    public function create(array $data)
    {
        return Task::create($data);
    }
}
```

1.3 TaskService.php

```
<?php
namespace App\Services;

use App\Repositories\TaskRepositoryInterface;

class TaskService
{
    protected $taskRepository;

    public function __construct(TaskRepositoryInterface $taskRepository)
    {
        $this->taskRepository = $taskRepository;
    }

    public function listTasks()
```

```
    {
        return $this->taskRepository->getAll();
    }

    public function addTask(array $data)
    {
        // Business Logic: ตั้งค่า status เริ่มต้น
        if (!isset($data['status'])) {
            $data['status'] = 'pending';
        }
        return $this->taskRepository->create($data);
    }
}
```

1.4 TaskController.php

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

use App\Services\TaskService;
use Illuminate\Http\Request;

class TaskController extends Controller
{
    protected $taskService;

    public function __construct(TaskService $taskService)
    {
        $this->taskService = $taskService;
    }

    public function index()
    {
        $tasks = $this->taskService->listTasks();
        return view('tasks', compact('tasks'));
    }
```

```
    public function store(Request $request)
    {
        $data = $request->validate([
            'title' => 'required|string',
        ]);

        $this->taskService->addTask($data);
        return redirect()->route('tasks.index')->with('success', 'Task added!');
    }
}
```

1.5 Blade View tasks.blade.php

```
<!DOCTYPE html>
<html>
<head><title>Tasks</title></head>
<body>
<h1>Task List</h1>

@if(session('success'))
    <p style="color:green">{{ session('success') }}</p>
@endif

<form method="POST" action="{{ route('tasks.store') }}">
    @csrf
    <input type="text" name="title" placeholder="New Task" required>
    <button type="submit">Add Task</button>
</form>

<ul>
@foreach ($tasks as $task)
    <li>{{ $task->title }} ({{ $task->status }})</li>
@endforeach
</ul>
</body>
```

```
</html>
```

1.6 Routes (web.php)

```
use App\Http\Controllers\TaskController;

Route::get('/tasks', [TaskController::class, 'index'])->name('tasks.index');
Route::post('/tasks', [TaskController::class, 'store'])->name('tasks.store');
```

---

2. ตัวอย่างโปรแกรม Order Management

(สร้าง Service จัดการสถานะ order, คำนวณราคาทั้งหมด)

OrderService.php

```
<?php
namespace App\Services;

use App\Repositories\OrderRepositoryInterface;

class OrderService
{
    protected $orderRepository;

    public function __construct(OrderRepositoryInterface $orderRepository)
    {
        $this->orderRepository = $orderRepository;
    }

    public function createOrder(array $data)
    {
        // Business Logic: คำนวณราคาทั้งหมด
        $data['total_price'] = array_sum(array_column($data['items'], 'price'));

        return $this->orderRepository->create($data);
    }
}
```

---

3. ตัวอย่างโปรแกรม User Registration

(จัดการลงทะเบียนผู้ใช้ พร้อมตรวจสอบและเข้ารหัสรหัสผ่าน)

UserService.php

```
<?php
namespace App\Services;

use App\Repositories\UserRepositoryInterface;
use Illuminate\Support\Facades\Hash;

class UserService
{
    protected $userRepository;

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

    public function registerUser(array $data)
    {
        // Business Logic: เข้ารหัสรหัสผ่าน
        $data['password'] = Hash::make($data['password']);

        return $this->userRepository->create($data);
    }
}
```

ตัวอย่างแนวประยุกต์ 3 โปรแกรม

---

1. ระบบจองห้องประชุม (Booking Room)

- Service ตรวจสอบวันเวลาว่างก่อนบันทึก
- Repository จัดการตาราง booking ในฐานข้อมูล
- Controller รับ request และส่งต่อ Service

---

2. ระบบแจ้งเตือนอีเมล (Email Notification)