Premium

Laravel MVC Web Programming: Intermediate

(Integrative-Generative AI Edition)

Student Price Book Center

คำนำ

การพัฒนาเว็บแอปพลิเคชันในปัจจุบันไม่เพียงแต่ต้องการความสามารถในการสร้างฟังก์ชันพื้นฐานเท่านั้น แต่ยังต้องคำนึงถึงประสิทธิภาพ ความปลอดภัย ความยืดหยุ่น และโครงสร้างที่สามารถปรับขยายได้ หนังสือ **Laravel MVC Web Programming: Intermediate** เล่มนี้ถูกสร้างขึ้นเพื่อเป็นคู่มือในการพัฒนาทักษะจากระดับพื้นฐานสู่ระดับกลาง ช่วยให้ผู้อ่านเข้าใจองค์ประกอบสำคัญของการพัฒนาเว็บแอปพลิเคชันด้วย Laravel อย่างเป็นระบบ และสามารถประยุกต์ใช้ในโครงการจริงได้อย่างมั่นใจ

ใน **บทที่ 5: การจัดการฐานข้อมูลด้วย Eloquent ORM (Eloquent ORM Database Management)** หน้า 1 ผู้อ่านจะได้เรียนรู้วิธีการเชื่อมต่อฐานข้อมูลกับ Laravel อย่างมืออาชีพ ผ่านการสร้าง **Model** การใช้งาน **Migration** เพื่อสร้างตาราง และการจัดการข้อมูลด้วย **Eloquent CRUD (Create, Read, Update, Delete)** นอกจากนี้ยังแนะนำการใช้ **Query Builder** เบื้องต้นเพื่อเพิ่มความยืดหยุ่นในการทำงานกับข้อมูล รวมถึงการสร้างความสัมพันธ์ระหว่าง Model (Eloquent Relationships) เช่น One-to-One, One-to-Many และ Many-to-Many เพื่อให้การออกแบบฐานข้อมูลเป็นไปอย่างมีประสิทธิภาพ

บทที่ 6: Authentication และ Authorization (Authentication and Authorization) หน้า 107 จะพาผู้อ่านเข้าสู่หัวข้อที่สำคัญสำหรับการรักษาความปลอดภัยและการจัดการสิทธิ์การเข้าถึง โดยอธิบายการติดตั้งและใช้งาน **Laravel Breeze** หรือ **Jetstream** สำหรับการสร้างระบบ Authentication เบื้องต้น การลงทะเบียนและล็อกอินผู้ใช้ ตลอดจนการจัดการ **Role และ Permission** เพื่อควบคุมการเข้าถึงฟังก์ชันหรือข้อมูลต่าง ๆ ซึ่งถือเป็นส่วนสำคัญของเว็บแอปพลิเคชันที่มีผู้ใช้หลายกลุ่ม

ใน **บทที่ 7: การจัดการ Session และ Flash Messages (Session and Flash Messages Management)** หน้า 166 ผู้อ่านจะได้ทำความเข้าใจวิธีจัดการ **Session** ใน Laravel เพื่อเก็บข้อมูลสำคัญชั่วคราว การเรียกใช้ **Flash Messages** เพื่อสื่อสารผลลัพธ์การทำงานกับผู้ใช้ และการทำงานกับ **Cookies** ซึ่งเป็นเครื่องมือสำคัญในการสร้างประสบการณ์ผู้ใช้ที่ดีและการเก็บข้อมูลเพื่อประโยชน์ในการโต้ตอบระหว่างผู้ใช้กับระบบ

บทที่ 8: การจัดการไฟล์และอัปโหลดไฟล์ (File and File Upload Management) หน้า 212 จะสอนวิธีจัดการไฟล์อย่างปลอดภัยและมีประสิทธิภาพ เริ่มตั้งแต่การสร้างฟอร์มอัปโหลดไฟล์ การบันทึกไฟล์ลง **Storage** การใช้งาน **Laravel Filesystem** ทั้งในรูปแบบ Local และ S3 รวมถึงการตรวจสอบชนิดไฟล์และขนาดไฟล์ เพื่อให้มั่นใจว่าการจัดการไฟล์ในระบบเป็นไปตามมาตรฐานความปลอดภัย

หนังสือเล่มนี้เน้นการอธิบายอย่างละเอียดเป็นขั้นตอน พร้อมตัวอย่างโค้ดที่สามารถนำไปใช้งานได้จริงในสถานการณ์ต่าง ๆ นอกจากนี้ยังให้ความสำคัญกับแนวคิดเบื้องหลังแต่ละฟังก์ชัน เพื่อให้ผู้อ่านเข้าใจทั้งกระบวนการและเหตุผลในการเลือกใช้เทคนิคต่าง ๆ

ผู้เขียนเชื่อว่าด้วยเนื้อหาที่ครอบคลุมและการอธิบายที่เป็นระบบ หนังสือเล่มนี้จะช่วยให้ผู้อ่านที่มีพื้นฐาน Laravel อยู่แล้วสามารถต่อยอดทักษะและยกระดับความสามารถในการพัฒนาเว็บแอปพลิเคชันได้อย่างมั่นคง อีกทั้งยังเป็นก้าวสำคัญก่อนการเรียนรู้หัวข้อขั้นสูงในระดับ Advance ต่อไป

สุดท้ายนี้ ผู้เขียนขอขอบคุณผู้อ่านทุกท่านที่ให้ความสนใจในหนังสือชุด Laravel MVC Web Programming และหวังว่าหนังสือเล่มนี้จะเป็นเพื่อนคู่คิดที่ช่วยให้คุณสร้างสรรค์ผลงานเว็บแอปพลิเคชันที่มีคุณภาพ ตอบโจทย์การใช้งานจริง และก้าวสู่ความเป็นมืออาชีพในสายงานพัฒนาเว็บอย่างแท้จริง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 5
การจัดการฐานข้อมูลด้วย Eloquent ORM
(Eloquent ORM Database Management)

เนื้อหา

- การจัดการฐานข้อมูลด้วย Eloquent ORM
- การจัดการฐานข้อมูลด้วย Eloquent ORM (เชิงลึก)
- การตั้งค่าเชื่อมต่อฐานข้อมูลใน Laravel
- การสร้าง Model และการใช้ Migration สร้างตาราง (Laravel)
- การใช้งาน Eloquent CRUD (Create, Read, Update, Delete) ใน Laravel
- การใช้ Query Builder เบื้องต้นใน Laravel
- การสร้างความสัมพันธ์ระหว่าง Model ใน Laravel (Eloquent Relationships)

บทนำ

บทที่ 5: การจัดการฐานข้อมูลด้วย Eloquent ORM

การจัดการฐานข้อมูลเป็นหัวใจสำคัญของการพัฒนาเว็บแอปพลิเคชัน เนื่องจากข้อมูลเป็นองค์ประกอบหลักที่ระบบต้องจัดเก็บ ประมวลผล และแสดงผลให้กับผู้ใช้ Laravel มีเครื่องมือที่ช่วยให้การจัดการฐานข้อมูลทำได้อย่างสะดวกและเป็นระบบผ่าน **Eloquent ORM (Object-Relational Mapping)** ซึ่งออกแบบมาให้การทำงานกับฐานข้อมูลมีความเป็นเชิงวัตถุ (Object-Oriented) และอ่านง่ายกว่า SQL แบบดั้งเดิม

การเริ่มต้นใช้งานฐานข้อมูลใน Laravel ต้องทำการ **ตั้งค่าเชื่อมต่อฐานข้อมูล** ผ่านไฟล์ .env โดยกำหนดรายละเอียดสำคัญ เช่น ชื่อฐานข้อมูล ชื่อผู้ใช้ รหัสผ่าน และประเภทของฐานข้อมูล (MySQL, PostgreSQL, SQLite หรือ SQL Server) การตั้งค่าที่ถูกต้องทำให้ระบบสามารถเชื่อมต่อและดำเนินการกับฐานข้อมูลได้อย่างราบรื่น อีกทั้งยังสามารถปรับเปลี่ยนค่าได้ง่ายเมื่อย้ายระบบระหว่างสภาพแวดล้อมการพัฒนาและการใช้งานจริง

หลังจากตั้งค่าการเชื่อมต่อแล้ว ขั้นตอนต่อมาคือการ **สร้าง Model และการใช้ Migration** เพื่อสร้างตารางในฐานข้อมูล Model ทำหน้าที่แทนตารางในฐานข้อมูล และช่วยให้การติดต่อกับข้อมูลเป็นเชิงวัตถุ ส่วน Migration เป็นเครื่องมือที่ช่วยจัดการโครงสร้างตาราง เช่น การสร้างตารางใหม่ การเพิ่มคอลัมน์ หรือการแก้ไขคอลัมน์เดิม ซึ่งทำให้การพัฒนาระบบร่วมกันเป็นทีมทำได้ง่ายและเป็นมาตรฐาน

Eloquent ORM ช่วยให้การทำงานกับข้อมูลในรูปแบบ **CRUD (Create, Read, Update, Delete)** มีความสะดวกและอ่านง่าย เช่น การเพิ่มข้อมูลใหม่ด้วย create() หรือ save() การอ่านข้อมูลด้วย all() หรือ find() การอัปเดตข้อมูลด้วย update() และการลบข้อมูลด้วย delete() ซึ่งการเรียกใช้เมธอดเหล่านี้จะถูกแปลงเป็นคำสั่ง SQL โดยอัตโนมัติ ทำให้นักพัฒนาสามารถโฟกัสกับตรรกะการทำงานของแอปพลิเคชันมากกว่าการเขียนคำสั่ง SQL โดยตรง

นอกจากการใช้ Eloquent แล้ว Laravel ยังมี **Query Builder** ซึ่งเป็นอีกวิธีหนึ่งในการดึงหรือประมวลผลข้อมูลแบบยืดหยุ่นและควบคุมการเขียน Query ได้ละเอียดขึ้น Query Builder รองรับทั้งการเลือกข้อมูล การกรอง การเรียงลำดับ และการเชื่อมตาราง ทำให้นักพัฒนาสามารถใช้วิธีนี้ได้เมื่อการประมวลผลข้อมูลซับซ้อนเกินกว่าจะใช้ Eloquent เพียงอย่างเดียว

อีกคุณสมบัติสำคัญของ Eloquent คือการ **สร้างความสัมพันธ์ระหว่าง Model** ซึ่งช่วยให้การทำงานกับข้อมูลที่เชื่อมโยงกันหลายตารางมีความสะดวก ตัวอย่างเช่น ความสัมพันธ์แบบ **One-to-One** สำหรับข้อมูลที่จับคู่กันหนึ่งต่อหนึ่ง **One-to-Many** สำหรับข้อมูลที่เชื่อมโยงแบบหนึ่งต่อหลาย และ **Many-to-Many** สำหรับข้อมูลที่มีความสัมพันธ์ซับซ้อนหลายต่อหลาย การกำหนดความสัมพันธ์นี้ช่วยให้การดึงข้อมูลที่เกี่ยวข้องทำได้ง่ายขึ้นเพียงการเรียกเมธอดที่กำหนดไว้ใน Model

บทนี้จึงถือเป็นพื้นฐานสำคัญสำหรับการพัฒนาเว็บแอปพลิเคชันที่ต้องทำงานกับฐานข้อมูล เนื้อหาตั้งแต่การตั้งค่าเชื่อมต่อฐานข้อมูล การสร้าง Model และ Migration การใช้ Eloquent สำหรับ CRUD การใช้ Query Builder เบื้องต้น ตลอดจนการสร้างความสัมพันธ์ระหว่าง Model จะช่วยให้นักพัฒนาสามารถจัดการข้อมูลได้อย่างเป็นระบบ ยืดหยุ่น และรองรับการขยายตัวของระบบในอนาคตได้อย่างมีประสิทธิภาพ

การจัดการฐานข้อมูลด้วย Eloquent ORM

- การตั้งค่าเชื่อมต่อฐานข้อมูล
- การสร้าง Model และการใช้ Migration สร้างตาราง
- การใช้งาน Eloquent CRUD (Create, Read, Update, Delete)
- การใช้ Query Builder เบื้องต้น
- การสร้างความสัมพันธ์ระหว่าง Model (One-to-One, One-to-Many, Many-to-Many)

---

1. การตั้งค่าเชื่อมต่อฐานข้อมูล

- เปิดไฟล์ .env ในโฟลเดอร์โปรเจกต์ Laravel
- กำหนดค่าการเชื่อมต่อฐานข้อมูล เช่น

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=your_database_name
```

```
DB_USERNAME=your_username
DB_PASSWORD=your_password
```

- Laravel จะใช้ข้อมูลนี้ในการเชื่อมต่อฐานข้อมูลผ่าน config/database.php
- สามารถเลือกฐานข้อมูลได้หลายชนิด เช่น MySQL, PostgreSQL, SQLite, SQL Server

---

2. การสร้าง Model และการใช้ Migration สร้างตาราง

- สร้าง Migration และ Model พร้อมกันโดยใช้คำสั่ง Artisan CLI:

```
php artisan make:model Post -m
```

- ตัวอย่างไฟล์ Migration (database/migrations/xxxx_xx_xx_create_posts_table.php)

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('title');
        $table->text('content');
        $table->timestamps();
    });
}
```

- รัน Migration เพื่อสร้างตารางในฐานข้อมูล

```
php artisan migrate
```

- Model (app/Models/Post.php) จะถูกสร้างขึ้นโดยอัตโนมัติ:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $fillable = ['title', 'content'];
}
```

- protected $fillable กำหนดฟิลด์ที่สามารถ Mass Assign ได้

---

3. การใช้งาน Eloquent CRUD (Create, Read, Update, Delete)

สร้างข้อมูล (Create)

```
$post = Post::create([
```

```
    'title' => 'My First Post',
    'content' => 'This is the content of my first post.',
]);
```

อ่านข้อมูล (Read)

```
$posts = Post::all(); // ดึงข้อมูลทั้งหมด
$post = Post::find(1); // ดึงข้อมูล ID = 1
```

แก้ไขข้อมูล (Update)

```
$post = Post::find(1);
$post->title = 'Updated Title';
$post->save();
```

ลบข้อมูล (Delete)

```
$post = Post::find(1);
$post->delete();
```

---

4. การใช้ Query Builder เบื้องต้น

- Query Builder ช่วยสร้าง SQL Query ได้ง่าย เช่น

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->where('status', 'active')->get();
```

- ตัวอย่าง Query Builder

```
$posts = DB::table('posts')
    ->where('title', 'like', '%Laravel%')
    ->orderBy('created_at', 'desc')
    ->limit(5)
    ->get();
```

- Query Builder สามารถใช้งานได้พร้อม Eloquent หรือแยกต่างหาก

---

5. การสร้างความสัมพันธ์ระหว่าง Model

One-to-One (เช่น User มี Profile หนึ่งอัน)

Model User

```
public function profile()
{
    return $this->hasOne(Profile::class);
```

```
}
```

Model Profile

```
public function user()
{
    return $this->belongsTo(User::class);
}
```

One-to-Many (เช่น Post มีหลาย Comment)

Model Post

```
public function comments()
{
    return $this->hasMany(Comment::class);
}
```

Model Comment

```
public function post()
{
    return $this->belongsTo(Post::class);
}
```

Many-to-Many (เช่น User มีหลาย Role และ Role มีหลาย User)

Model User

```
public function roles()
{
    return $this->belongsToMany(Role::class);
}
```

Model Role

```
public function users()
{
    return $this->belongsToMany(User::class);
}
```

- ต้องมีตาราง pivot (เช่น role_user) ที่เก็บความสัมพันธ์

การจัดการฐานข้อมูลด้วย Eloquent ORM (เชิงลึก)

---

1. การตั้งค่าเชื่อมต่อฐานข้อมูล

- Laravel ใช้ไฟล์ .env เป็นตัวกำหนดคอนฟิกสำหรับการเชื่อมต่อฐานข้อมูล
- โดยค่าเริ่มต้น DB_CONNECTION จะเป็น mysql แต่สามารถเปลี่ยนเป็น pgsql, sqlite หรือ sqlsrv ได้ตามฐานข้อมูลที่ใช้งาน
- **เหตุผลที่ใช้ .env** คือเพื่อความปลอดภัย ไม่เก็บข้อมูลสำคัญในโค้ด และสามารถเปลี่ยนแปลงได้ง่ายในแต่ละ environment (dev, staging, production)
- ตัวอย่าง .env:
- DB_CONNECTION=mysql
- DB_HOST=127.0.0.1
- DB_PORT=3306
- DB_DATABASE=laravel_db
- DB_USERNAME=root
- DB_PASSWORD=secret
- สามารถรันคำสั่ง php artisan config:cache เพื่อ cache config และเพิ่มประสิทธิภาพใน production

---

2. การสร้าง Model และ Migration

- **Model** คือคลาสที่แทนตารางในฐานข้อมูล และทำหน้าที่ติดต่อกับข้อมูลในตารางนั้น
- ใช้คำสั่ง php artisan make:model ModelName -m เพื่อสร้าง Model และ Migration พร้อมกัน
- **Migration** คือไฟล์ PHP ที่ใช้จัดการสร้าง/แก้ไขโครงสร้างตารางในฐานข้อมูลอย่างเป็นระบบ
- การใช้ Migration ช่วยให้การสร้างตารางเป็น version control และง่ายต่อการ deploy
- Migration มีสอง method สำคัญ:
    - up() — ใช้สำหรับสร้างหรือแก้ไขตาราง
    - down() — ใช้สำหรับย้อนกลับการเปลี่ยนแปลง (rollback)
- ตัวอย่างโครงสร้าง Migration:
- public function up()
- {
- Schema::create('users', function (Blueprint $table) {
- $table->id();
- $table->string('name');
- $table->string('email')->unique();
- $table->timestamps();
- });
- }

- **การรัน migration:** php artisan migrate จะตรวจสอบ migration ทั้งหมดที่ยังไม่รันและดำเนินการสร้างตาราง

---

3. การใช้งาน Eloquent CRUD

- Eloquent ORM ทำให้การติดต่อฐานข้อมูลเป็นแบบ Object-Oriented สะดวกมากขึ้น

Create (สร้างข้อมูล)

- สามารถใช้ Model::create() โดยต้องระบุฟิลด์ที่อนุญาตให้ Mass Assignment ($fillable หรือ $guarded)
- หรือสร้าง instance แล้ว save() เช่น
- $user = new User();
- $user->name = 'John';
- $user->email = 'john@example.com';
- $user->save();

Read (อ่านข้อมูล)

- ใช้ Model::all(), Model::find(id), หรือ Model::where() เพื่อดึงข้อมูลตามเงื่อนไข

Update (แก้ไขข้อมูล)

- หา record ที่ต้องการ แล้วเปลี่ยนแปลงค่า จากนั้นเรียก save()

Delete (ลบข้อมูล)

- หา record แล้วเรียก delete() เพื่อถอดข้อมูลออก
- **หมายเหตุ:** มีเมธอดอื่น ๆ เช่น update(), destroy() ที่ช่วยให้เขียนโค้ดสั้นและชัดเจนขึ้น

---

4. การใช้ Query Builder เบื้องต้น

- Query Builder คือ API ที่ช่วยเขียน SQL query แบบ fluent syntax และรองรับฐานข้อมูลหลายชนิด
- ใช้เมื่อไม่ต้องการ Model หรือ Eloquent (เช่น query ที่ซับซ้อน หรือประสิทธิภาพสูง)
- ตัวอย่าง:
- $users = DB::table('users')->where('active', 1)->orderBy('created_at', 'desc')->get();
- Query Builder สามารถทำงานร่วมกับ Eloquent ได้ เช่นใช้ใน scope หรือ subquery
- รองรับ join, groupBy, having, paginate เป็นต้น

---

5. การสร้างความสัมพันธ์ระหว่าง Model

การจัดการความสัมพันธ์ของข้อมูลในฐานข้อมูลผ่าน Eloquent เป็นจุดเด่นที่สำคัญ ช่วยให้เขียนโค้ดจัดการข้อมูลที่เกี่ยวข้องกันง่ายและชัดเจน

One-to-One

- เช่น User มี Profile หนึ่งเรคอร์ด

```
// User.php
public function profile()
{
    return $this->hasOne(Profile::class);
}

// Profile.php
public function user()
{
    return $this->belongsTo(User::class);
}
```

- การเข้าถึง: $user->profile หรือ $profile->user
- โดยตาราง Profile ต้องมี user_id เป็น foreign key

One-to-Many

- เช่น Post มีหลาย Comment

```
// Post.php
public function comments()
{
    return $this->hasMany(Comment::class);
}

// Comment.php
public function post()
{
    return $this->belongsTo(Post::class);
}
```

- การเข้าถึง: $post->comments จะได้ Collection ของ Comment ทั้งหมด

Many-to-Many

- เช่น User มีหลาย Role และ Role มีหลาย User

```
// User.php
public function roles()
```

```
{
    return $this->belongsToMany(Role::class);
}

// Role.php
public function users()
{
    return $this->belongsToMany(User::class);
}
```

- ต้องมีตาราง Pivot เช่น role_user ที่เก็บ user_id และ role_id
- สามารถใช้ attach(), detach(), sync() เพื่อจัดการความสัมพันธ์

สรุป

หัวข้อ	รายละเอียดเชิงลึก
การตั้งค่า DB	กำหนดใน .env ค่าต่าง ๆ เช่น host, username, password, dbname
Model & Migration	Model ใช้แทนตาราง DB, Migration สร้าง/แก้ไขโครงสร้างตาราง
CRUD Eloquent	ใช้ method เช่น create(), find(), save(), delete()
Query Builder	API สำหรับเขียน SQL แบบ Fluent และใช้ได้ทุก DB
ความสัมพันธ์ Model	กำหนด relation แบบ hasOne, hasMany, belongsToMany, ใช้งานง่าย

การตั้งค่าเชื่อมต่อฐานข้อมูลใน Laravel

---

1. ไฟล์ .env — จุดเริ่มต้นของการตั้งค่า

- Laravel ใช้ไฟล์ .env เพื่อเก็บคอนฟิกที่เปลี่ยนแปลงได้ตาม environment (development, production ฯลฯ)
- ไฟล์นี้ไม่ควรถูก commit ขึ้นระบบ version control เพราะมีข้อมูลลับ เช่น รหัสผ่านฐานข้อมูล

ตัวอย่างการตั้งค่า database ใน .env:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_database
DB_USERNAME=root
```

DB_PASSWORD=secret

---

2. ความหมายของแต่ละตัวแปร

ตัวแปร	ความหมาย	ตัวอย่างค่า
DB_CONNECTION	ประเภทฐานข้อมูลที่ใช้ (ชื่อ driver)	mysql, pgsql, sqlite, sqlsrv
DB_HOST	ที่อยู่ server ของฐานข้อมูล	127.0.0.1, localhost, IP หรือ hostname
DB_PORT	พอร์ตเชื่อมต่อฐานข้อมูล	3306 (MySQL), 5432 (PostgreSQL)
DB_DATABASE	ชื่อฐานข้อมูลที่จะเชื่อมต่อ	laravel_database
DB_USERNAME	ชื่อผู้ใช้ฐานข้อมูล	root
DB_PASSWORD	รหัสผ่านของผู้ใช้ฐานข้อมูล	secret

---

3. ตัวอย่างการเชื่อมต่อฐานข้อมูลแต่ละประเภท

MySQL

DB_CONNECTION=mysql

DB_HOST=127.0.0.1

DB_PORT=3306

DB_DATABASE=laravel_db

DB_USERNAME=root

DB_PASSWORD=secret

PostgreSQL

DB_CONNECTION=pgsql

DB_HOST=127.0.0.1

DB_PORT=5432

DB_DATABASE=laravel_db

DB_USERNAME=postgres

DB_PASSWORD=secret

SQLite

- กำหนดไฟล์ฐานข้อมูลที่เก็บในโปรเจกต์ เช่น

DB_CONNECTION=sqlite

DB_DATABASE=/full/path/to/database.sqlite

- หรือใช้ไฟล์ในโปรเจกต์

DB_CONNECTION=sqlite

```
DB_DATABASE=database/database.sqlite
```

---

4. การตั้งค่าในไฟล์ config/database.php

- Laravel อ่านค่าใน .env แล้วนำมาใส่ใน config ที่อยู่ใน config/database.php
- ตัวอย่างส่วนที่เกี่ยวข้อง:

```
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'strict' => true,
    'engine' => null,
],
```

- env() จะอ่านค่าในไฟล์ .env โดยมีค่าดีฟอลต์ถ้าไม่เจอ

---

5. การทดสอบการเชื่อมต่อ

- สามารถใช้คำสั่ง Artisan เช่น

```
php artisan migrate:status
```

เพื่อทดสอบว่าการเชื่อมต่อฐานข้อมูลสำเร็จหรือไม่

- หรือเขียนโค้ดเช็คง่าย ๆ:

```
try {
    DB::connection()->getPdo();
    echo "Connected to database successfully.";
} catch (\Exception $e) {
    echo "Could not connect to the database. Please check your configuration.";
}
```

---

6. คำแนะนำเพิ่มเติม

- ใน environment production ให้ตั้งค่าการเชื่อมต่อให้ถูกต้อง ปลอดภัย และเก็บข้อมูลลับใน .env
- หลีกเลี่ยงการเก็บข้อมูลลับในไฟล์ config หรือโค้ดโดยตรง
- ควรตั้งค่าให้ DB_CONNECTION ตรงกับฐานข้อมูลที่ใช้งานจริง เช่น บางครั้ง dev ใช้ SQLite แต่ production ใช้ MySQL
- อย่าลืม restart server หรือ clear config cache (php artisan config:clear) หลังแก้ไข .env

---

นี่คือตัวอย่างโปรแกรม Laravel แบบเต็มไฟล์ 3 โปรแกรมพื้นฐาน และ 3 โปรแกรมแนวประยุกต์ ที่แสดงการตั้งค่าเชื่อมต่อฐานข้อมูลและใช้งาน Eloquent ORM พร้อมคำอธิบายโค้ดและผลการรัน

ตัวอย่างโปรแกรมพื้นฐาน 3 โปรแกรม

ตัวอย่างที่ 1: การเชื่อมต่อฐานข้อมูลและดึงข้อมูลด้วย Eloquent (Model: User)

โครงสร้างไฟล์

routes/web.php

app/Models/User.php

app/Http/Controllers/UserController.php

resources/views/users.blade.php

---

routes/web.php

```
use App\Http\Controllers\UserController;

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

---

app/Models/User.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $fillable = ['name', 'email'];
}
```

---

app/Http/Controllers/UserController.php

```
namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    public function index()
    {
        $users = User::all();
        return view('users', compact('users'));
    }
}
```

---

resources/views/users.blade.php

```
<!DOCTYPE html>
<html>
<head><title>User List</title></head>
<body>

<h1>Users</h1>

<ul>
@foreach($users as $user)
    <li>{{ $user->name }} - {{ $user->email }}</li>
@endforeach
</ul>

</body>
</html>
```

คำอธิบาย

- เชื่อมต่อฐานข้อมูลผ่าน .env (เช่น MySQL)
- Model User แทนตาราง users

- Controller ดึงข้อมูลทั้งหมดแล้วส่งไปที่ View
- View แสดงรายชื่อผู้ใช้ในรูปแบบลิสต์

ผลการรัน

แสดงหน้าเว็บรายชื่อผู้ใช้ทั้งหมดในตาราง users

ตัวอย่างที่ 2: การสร้างข้อมูลใหม่ (Create) ด้วยฟอร์ม

โครงสร้างไฟล์

routes/web.php

app/Http/Controllers/ProductController.php

app/Models/Product.php

resources/views/product_create.blade.php

---

routes/web.php

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

Route::get('/product/create', [ProductController::class, 'create']);
Route::post('/product/store', [ProductController::class, 'store'])->name('product.store');
```

---

app/Models/Product.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    protected $fillable = ['name', 'price'];
}
```

---

app/Http/Controllers/ProductController.php

```
namespace App\Http\Controllers;

use App\Models\Product;
use Illuminate\Http\Request;
```

```
class ProductController extends Controller
{
    public function create()
    {
        return view('product_create');
    }

    public function store(Request $request)
    {
        $request->validate([
            'name' => 'required|min:3',
            'price' => 'required|numeric|min:0',
        ]);

        Product::create($request->all());

        return redirect()->back()->with('success', 'Product created successfully!');
    }
}
```

---

resources/views/product_create.blade.php

```
<!DOCTYPE html>
<html>
<head><title>Create Product</title></head>
<body>

<h1>Create Product</h1>

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

@if ($errors->any())
```

```
    <div style="color: red;">
        <ul>
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
        </ul>
    </div>
@endif

<form method="POST" action="{{ route('product.store') }}">
    @csrf
    <label>Name:</label><br>
    <input type="text" name="name" value="{{ old('name') }}"><br><br>

    <label>Price:</label><br>
    <input type="number" name="price" step="0.01" value="{{ old('price') }}"><br><br>

    <button type="submit">Create</button>
</form>

</body>
</html>
```

คำอธิบาย

- มีฟอร์มกรอกชื่อสินค้าและราคา
- Controller ตรวจสอบ Validation
- ถ้าผ่าน สร้างข้อมูลใหม่ในตาราง products
- แจ้งผลลัพธ์ผ่าน session

ผลการรัน

- แสดงฟอร์มสร้างสินค้า
- ถ้าข้อมูลไม่ถูกต้อง จะแสดงข้อความ error
- ถ้าถูกต้อง แสดงข้อความ success

ตัวอย่างที่ 3: การอัปเดตข้อมูล (Update)

โครงสร้างไฟล์

routes/web.php

app/Http/Controllers/PostController.php

app/Models/Post.php

resources/views/post_edit.blade.php

---

routes/web.php

```
use App\Http\Controllers\PostController;

Route::get('/post/{id}/edit', [PostController::class, 'edit'])->name('post.edit');
Route::post('/post/{id}/update', [PostController::class, 'update'])->name('post.update');
```

---

app/Models/Post.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $fillable = ['title', 'content'];
}
```

---

app/Http/Controllers/PostController.php

```
namespace App\Http\Controllers;

use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function edit($id)
    {
        $post = Post::findOrFail($id);
```

```
        return view('post_edit', compact('post'));
    }

    public function update(Request $request, $id)
    {
        $request->validate([
            'title' => 'required|min:3',
            'content' => 'required',
        ]);

        $post = Post::findOrFail($id);
        $post->update($request->all());

        return redirect()->route('post.edit', $post->id)->with('success', 'Post updated successfully!');
    }
}
```

---

resources/views/post_edit.blade.php

```
<!DOCTYPE html>
<html>
<head><title>Edit Post</title></head>
<body>

<h1>Edit Post</h1>

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

@if ($errors->any())
    <div style="color: red;">
        <ul>
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
```

```
        @endforeach
        </ul>
    </div>
@endif

<form method="POST" action="{{ route('post.update', $post->id) }}">
    @csrf

    <label>Title:</label><br>
    <input type="text" name="title" value="{{ old('title', $post->title) }}"><br><br>

    <label>Content:</label><br>
    <textarea name="content">{{ old('content', $post->content) }}</textarea><br><br>

    <button type="submit">Update</button>
</form>

</body>
</html>
```

คำอธิบาย

- ดึงข้อมูลโพสต์ที่จะแก้ไขมาแสดงในฟอร์ม
- ฟอร์มส่งข้อมูลไปอัปเดตใน Controller
- Controller ทำ Validation และอัปเดตข้อมูลในฐานข้อมูล
- แจ้งผลลัพธ์ผ่าน session

ผลการรัน

- แสดงฟอร์มแก้ไขโพสต์
- แสดงข้อความ error ถ้าข้อมูลไม่ถูกต้อง
- แสดงข้อความ success เมื่ออัปเดตสำเร็จ

ตัวอย่างโปรแกรมแนวประยุกต์ 3 โปรแกรม

ตัวอย่างที่ 4: ระบบ Blog Post กับความสัมพันธ์ One-to-Many (Post - Comments)

โครงสร้างไฟล์

routes/web.php

app/Models/Post.php

app/Models/Comment.php

app/Http/Controllers/PostController.php

resources/views/post_show.blade.php

---

routes/web.php

```
use App\Http\Controllers\PostController;

Route::get('/post/{id}', [PostController::class, 'show'])->name('post.show');
```

---

app/Models/Post.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    protected $fillable = ['title', 'content'];

    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

---

app/Models/Comment.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
```

```
    protected $fillable = ['post_id', 'author', 'comment'];

    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

---

app/Http/Controllers/PostController.php

```
namespace App\Http\Controllers;

use App\Models\Post;

class PostController extends Controller
{
    public function show($id)
    {
        $post = Post::with('comments')->findOrFail($id);
        return view('post_show', compact('post'));
    }
}
```

---

resources/views/post_show.blade.php

```
<!DOCTYPE html>
<html>
<head><title>Post Details</title></head>
<body>

<h1>{{ $post->title }}</h1>
<p>{{ $post->content }}</p>

<h3>Comments</h3>
<ul>
    @foreach($post->comments as $comment)
```

```
        <li><strong>{{ $comment->author }}</strong>: {{ $comment->comment }}</li>
    @endforeach
</ul>

</body>
</html>
```

คำอธิบาย

- แสดงข้อมูลโพสต์พร้อมคอมเมนต์ที่เกี่ยวข้องผ่านความสัมพันธ์
- ใช้ eager loading (with('comments')) เพื่อลดจำนวน query

ผลการรัน

- แสดงโพสต์พร้อมคอมเมนต์ในหน้าเดียวกัน

ตัวอย่างที่ 5: ระบบผู้ใช้และบทบาท (Roles) แบบ Many-to-Many

โครงสร้างไฟล์

routes/web.php

app/Models/User.php

app/Models/Role.php

app/Http/Controllers/UserController.php

resources/views/user_roles.blade.php

---

routes/web.php

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}/roles', [UserController::class, 'showRoles'])->name('user.roles');
```

---

app/Models/User.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
```

```
    protected $fillable = ['name', 'email'];

    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

---

app/Models/Role.php

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    protected $fillable = ['name'];

    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

---

app/Http/Controllers/UserController.php

```
namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    public function showRoles($id)
    {
        $user = User::with('roles')->findOrFail($id);
        return view('user_roles', compact('user'));
```