DESIGN THINKING PART 1: TẠI SAO & ĐỘNG CƠ

"Mỗi dòng code đều có lý do"

Lời mở đầu

Khi bắt đầu xây dựng Admin Panel cho CineBook, tôi không chỉ nghĩ về việc "làm cho nó chạy". Tôi nghĩ về **người sẽ dùng nó hàng ngày** - những admin viên của rạp phim.

Họ không phải là developer. Họ có thể là nhân viên quản lý 50 tuổi, quen với Excel hơn là web app. Họ cần một công cụ **đơn giản, hiệu quả, không gây nhầm lẫn**.

Đó là kim chỉ nam cho mọi quyết định thiết kế.

---

1. TẠI SAO CHỌN MVC PATTERN?

Vấn đề tôi gặp phải

Ban đầu, tôi viết code theo kiểu "spaghetti" - logic xử lý trộn lẫn với hiển thị. Một file PHP vừa query database, vừa xử lý dữ liệu, vừa render HTML.

Kết quả?

- Khó debug: Không biết lỗi ở đâu
- Khó maintain: Sửa một chỗ, hỏng chỗ khác
- Khó mở rộng: Thêm tính năng = viết lại từ đầu

Quyết định: Tách biệt trách nhiệm

MVC không phải vì "ai cũng dùng". MVC vì nó **giải quyết vấn đề thực tế**:

```
Trước MVC:
- 1 file movies.php = 500 dòng code
- Query + Logic + HTML lẫn lộn
- Developer A sửa → Developer B khóc

Sau MVC:
- Controller: 100 dòng (xử lý logic)
- Model: 50 dòng (làm việc với database)
- View: 200 dòng (hiển thị UI)
- 3 người có thể làm việc song song
```

Bài học rút ra

> **"Tách biệt không phải để phức tạp hóa, mà để đơn giản hóa việc hiểu code."**

Khi đọc `AdminMovieController@store`, tôi biết ngay đây là nơi xử lý việc tạo phim mới. Không cần tìm trong 500 dòng code.

---

2. TẠI SAO TÁCH RIÊNG ADMIN VÀ USER?

Cân nhắc ban đầu

Có 2 hướng tiếp cận:

1. **Một giao diện chung**: Admin và User dùng chung, chỉ khác quyền
2. **Hai giao diện riêng**: Admin có layout và flow hoàn toàn khác

Tại sao tôi chọn hướng 2?

Lý do 1: Mục đích sử dụng khác nhau

```
User muốn:
- Xem phim nhanh
- Đặt vé dễ dàng
- Trải nghiệm giải trí

Admin muốn:
- Quản lý hiệu quả
- Nhìn thấy số liệu
- Thao tác hàng loạt
```

Thiết kế cho 2 mục đích khác nhau bằng 1 giao diện = không ai hài lòng.

Lý do 2: Bảo mật

```
Nếu dùng chung layout:
- URL: /movies (cả user và admin)
- Kiểm tra role trong view
- Dễ sót, dễ lộ chức năng admin

Nếu tách riêng:
- URL: /admin/movies (chỉ admin)
- Middleware chặn từ đầu
- Không thể truy cập nếu không có quyền
```

Lý do 3: Dễ phát triển độc lập

Team A làm trang User (focus: UX đẹp, marketing) Team B làm trang Admin (focus: hiệu quả, dữ liệu) Không đụng chạm nhau.

Kết quả

```php
// Route Group với prefix và middleware
Route::prefix('admin')
    ->middleware(['auth', 'role:admin'])
    ->group(function () {
        // Tất cả route admin ở đây
        // An toàn, tách biệt, dễ quản lý
    });
```

---

3. TẠI SAO DÙNG MIDDLEWARE CHO PHÂN QUYỀN?

Các cách làm khác tôi đã cân nhắc

Cách 1: Kiểm tra trong Controller

```php
public function index() {
    if (auth()->user()->role !== 'admin') {
        abort(403);
    }
    // Logic...
}
```

Vấn đề: Phải viết đi viết lại ở MỌI function.

Cách 2: Kiểm tra trong View

```php
@if(auth()->user()->role === 'admin')
    <button>Delete</button>
@endif
```

Vấn đề: User vẫn có thể gọi trực tiếp URL.

Quyết định: Middleware

Middleware như một **bảo vệ đứng ở cổng**. Không cần kiểm tra ở trong nhà nữa.

```php
// CheckRole.php
public function handle($request, Closure $next, $role)
{
    if (auth()->user()->role !== $role) {
        return redirect('/')->with('error', 'Access denied');
    }
    return $next($request);
}
```

Động cơ sâu xa

> **"Bảo mật tốt nhất là bảo mật mà developer không thể quên."**

Với middleware:

- Thêm route mới? Tự động được bảo vệ (nếu trong group)
- Developer mới join? Không cần nhớ check quyền
- Code review? Chỉ cần xem route file

---

4. TẠI SAO CHỌN DATABASE TRANSACTION CHO ROOM + SEATS?

Tình huống thực tế đã gặp

Khi tạo phòng chiếu 100 ghế:

1. Tạo Room thành công
2. Tạo được 50 ghế
3. Lỗi network/server
4. **Kết quả**: Phòng có 50 ghế "ma"

Giải pháp: All or Nothing

```
DB::transaction(function () {
    $room = Room::create([...]);

    foreach ($seatLayout as $seat) {
        Seat::create([
            'room_id' => $room->id,
            ...
        ]);
    }
});
// Nếu có lỗi → rollback TOÀN BỘ
// Không có phòng nửa vời
```

Nguyên tắc áp dụng

Transaction được dùng ở những nơi:

- Tạo phòng + ghế
- Đặt vé + chọn ghế
- Hủy vé + giải phóng ghế

> **"Không có trạng thái nửa vời. Hoặc hoàn thành, hoặc như chưa bắt đầu."**

---

5. TẠI SAO THIẾT KẾ QR CODE NHƯ VẬY?

Yêu cầu thực tế

Rạp phim cần:

- Vé điện tử thay vì vé giấy
- Check-in nhanh (< 3 giây)
- Chống giả mạo
- Không cần internet khi scan (lý tưởng)

Quyết định thiết kế

Không dùng QR chứa booking_id đơn thuần

Vì sao? Dễ đoán. `booking_id = 123` → Hacker thử `124, 125,...`

Dùng SHA-256 hash

```
$qrCode = hash('sha256', $bookingId . $seatId . config('app.key'));
```

Kết quả: `a1b2c3d4e5f6...` - Không thể đoán được

Trade-off tôi chấp nhận

```
Ưu điểm:
- Bảo mật cao
- Không thể giả mạo
- Mỗi ghế một QR riêng

Nhược điểm:
- Cần database lookup khi scan
- Không offline được 100%
```

Với quy mô rạp phim nhỏ-vừa, database lookup là chấp nhận được.

---

6. TẠI SAO KHÔNG DÙNG SOFT DELETE?

Soft Delete là gì?

Thay vì xóa thật, đánh dấu `deleted_at = now()`. Dữ liệu vẫn còn trong database.

Tôi KHÔNG dùng ở đâu và TẠI SAO

Không dùng cho Seats:

```
Lý do: Seat không có ý nghĩa khi tách khỏi Room
Nếu Room bị xóa → Seats cũng nên xóa thật
```

Không dùng cho Showtimes cũ:

```
Lý do: Suất chiếu đã qua = lịch sử
Cần giữ để báo cáo, không cần "khôi phục"
Giải pháp: Flag status = 'completed', không phải soft delete
```

Tôi CÓ dùng ở đâu

Dùng cho Users:

```
Lý do: User có thể yêu cầu khôi phục tài khoản
Lịch sử booking cần giữ liên kết với user
```

Dùng cho Movies:

```
Lý do: Phim hết chiếu có thể chiếu lại
Showtimes cũ reference đến movie
```

Nguyên tắc

> **"Soft delete khi cần khôi phục hoặc cần giữ reference. Hard delete khi dữ liệu mất ý nghĩa."**

---

7. TẠI SAO DÙNG EAGER LOADING?

Vấn đề N+1 Query

```php
// Code ban đầu
$bookings = Booking::all(); // 1 query

foreach ($bookings as $booking) {
    echo $booking->user->name;    // N queries
    echo $booking->showtime->movie->title; // N queries nữa
}
// Tổng: 1 + N + N = quá nhiều!
```

Với 100 bookings → 201 queries. Database khóc.

Giải pháp

```php
$bookings = Booking::with(['user', 'showtime.movie'])->get();
// Chỉ 3 queries cho mọi dữ liệu
```

Tại sao không dùng JOIN?

```
Eager Loading:
- Code sạch, dễ đọc
- Laravel tự optimize
- Trả về Eloquent objects

Raw JOIN:
- Code dài, khó maintain
- Phải map thủ công
- Dễ sai khi table thay đổi
```

Tôi ưu tiên **maintainability** hơn **micro-optimization**.

---

8. TẠI SAO DASHBOARD HIỂN THỊ NHỮNG SỐ LIỆU ĐÓ?

Suy nghĩ về người dùng

Admin mở Dashboard lúc 8h sáng muốn biết gì?

```
Câu hỏi trong đầu admin:
1. "Hôm qua bán được bao nhiêu?" → Doanh thu
2. "Có bao nhiêu vé được đặt?" → Số booking
3. "Phim nào đang hot?" → Phim có nhiều đặt vé
4. "Hôm nay có suất nào?" → Lịch chiếu
```

Thiết kế theo thứ tự ưu tiên

```
Row 1: Số liệu quan trọng nhất (Cards)
- Tổng doanh thu
- Số booking
- Số user
- Số phim

Row 2: Biểu đồ xu hướng
- Doanh thu 7 ngày qua
- So sánh với tuần trước

Row 3: Chi tiết
- Phim hot
- Booking gần đây
```

Nguyên tắc

> **"Thông tin quan trọng nhất phải thấy được không cần scroll."**

---

9. TẠI SAO DÙNG FLASH MESSAGE?

Vấn đề UX

User click "Xóa phim":

- Không có thông báo gì
- Trang refresh
- User: "Đã xóa chưa? Hay bị lỗi?"

Quyết định

Mọi action quan trọng đều có feedback:

```
return redirect()->route('admin.movies.index')
    ->with('success', 'Xóa phim thành công!');
```

Thiết kế message

```
Success (xanh): Hành động hoàn tất
- "Tạo phim thành công"
- "Cập nhật suất chiếu thành công"

Error (đỏ): Có lỗi xảy ra
- "Không thể xóa phòng đang có suất chiếu"
- "Email đã tồn tại"

Warning (vàng): Cảnh báo
- "Bạn có chắc muốn xóa?"

Info (xanh dương): Thông tin
- "Đang xử lý..."
```

Nguyên tắc

> **"Đừng để user đoán. Nói cho họ biết chuyện gì đã xảy ra."**

---

10. TẠI SAO VALIDATE Ở CẢ CLIENT VÀ SERVER?

Không chỉ validate ở client

```
// Client validation
if (title.length === 0) {
```

```
    showError('Tiêu đề không được trống');
    return;
}
```

User thấy lỗi ngay → UX tốt. Nhưng... user có thể tắt JavaScript hoặc gửi request trực tiếp.

Không chỉ validate ở server

```
// Server validation
$validated = $request->validate([
    'title' => 'required|max:255',
]);
```

Bảo mật → Data integrity đảm bảo. Nhưng... user phải đợi server response mới biết lỗi.

Quyết định: CẢ HAI

```
Client: UX tốt, feedback nhanh
Server: Bảo mật, source of truth

Client validation = "Courtesy check"
Server validation = "Enforcement"
```

Nguyên tắc

> **"Không tin client. Không làm khó user."**

---

11. TẠI SAO CHỌN CẤU TRÚC URL NHƯ VẬY?

RESTful thinking

```
/admin/movies          → Danh sách phim
/admin/movies/create   → Form tạo mới
/admin/movies/{id}     → Chi tiết một phim
/admin/movies/{id}/edit → Form sửa

Không phải:
/admin/getAllMovies
/admin/createNewMovie
/admin/getMovieById?id=123
```

Lý do

1. **Predictable**: Biết URL là biết được action

2. **Bookmarkable**: User có thể lưu link
3. **SEO friendly**: Dù admin không cần SEO
4. **Industry standard**: Developer mới hiểu ngay

Ngoại lệ

```
/admin/qr-checkin → Không phải CRUD thuần túy
/admin/dashboard  → Không có resource cụ thể
```

> **"Theo convention khi có thể. Phá lệ khi có lý do."**

---

12. TẠI SAO CHỌN NHỮNG MÀU SẮC ĐÓ?

Không phải chọn random

```
Primary (Blue): #3B82F6
- Tin cậy, chuyên nghiệp
- Dễ nhìn lâu không mỏi mắt
- Phù hợp dashboard business

Success (Green): #10B981
- Liên tưởng: Đúng, hoàn thành
- Universal understanding

Danger (Red): #EF4444
- Cảnh báo, xóa, lỗi
- Gây chú ý

Warning (Yellow): #F59E0B
- Cần xem xét, không nghiêm trọng
```

Accessibility

```
Contrast ratio > 4.5:1
- Text trên background phải đọc được
- Không chỉ dựa vào màu sắc
- Icon + Text + Color
```

Nguyên tắc

> **"Màu sắc không chỉ đẹp, màu sắc là ngôn ngữ."**

---

13. NHỮNG SAI LẦM VÀ BÀI HỌC

Sai lầm 1: Over-engineering ban đầu

```
Dự định: Tạo hệ thống permission phức tạp
- Role → Permission → Resource → Action
- 10 tables cho authorization

Thực tế: Chỉ cần 2 role
- Admin: Full access
- User: Customer access

Bài học: Giải quyết vấn đề thực tế, không phải vấn đề tưởng tượng
```

Sai lầm 2: Premature optimization

```
Dự định: Cache mọi thứ ngay từ đầu
- Redis cho sessions
- Memcached cho queries
- CDN cho static files

Thực tế: Rạp phim nhỏ, traffic thấp

Bài học: Optimize khi cần, measure trước khi optimize
```

Sai lầm 3: Quên mobile

```
Ban đầu: Design cho desktop
Kết quả: Admin dùng điện thoại check nhanh → UX tệ

Sửa: Responsive từ đầu

Bài học: Hỏi user họ dùng gì, đừng assume
```

---

14. NGUYÊN TẮC THIẾT KẾ CỐT LÕI

1. KISS - Keep It Simple, Stupid

```
Đơn giản không có nghĩa là thiếu tính năng
Đơn giản là: Mỗi tính năng dễ dùng
```

2. DRY - Don't Repeat Yourself

```
Không copy paste code
Tạo helper, component, trait khi cần reuse
```

3. YAGNI - You Aren't Gonna Need It

```
Không code tính năng "phòng khi cần"
Code khi có yêu cầu thực tế
```

4. Separation of Concerns

```
Mỗi phần code làm một việc
Controller không chứa business logic phức tạp
Model không chứa presentation logic
```

5. Fail Fast

```
Validate ngay khi nhận input
Throw exception rõ ràng
Không để lỗi lan ra xa
```

---

15. TÓM TẮT: TƯ DUY THIẾT KẾ

Trước khi code, tự hỏi:

1. **User là ai?** Admin 50 tuổi hay developer 25 tuổi?
2. **Họ muốn gì?** Quản lý hiệu quả hay trải nghiệm giải trí?
3. **Họ dùng trong hoàn cảnh nào?** Office với màn hình lớn hay on-the-go với điện thoại?
4. **Chuyện gì có thể sai?** Network lỗi? User nhập sai? Hacker tấn công?
5. **Làm sao để recover?** Transaction rollback? Error message? Retry mechanism?

Sau khi code, kiểm tra:

1. **Đọc lại được không?** 6 tháng sau có hiểu không?
2. **Test được không?** Có thể viết unit test không?
3. **Mở rộng được không?** Thêm tính năng có phải rewrite không?
4. **Secure không?** Có lỗ hổng nào không?

5. **Performant không?** Với 1000 users có chạy được không?

Kết luận

Mỗi dòng code trong Admin Panel đều có lý do tồn tại. Không phải vì "best practice nói vậy", mà vì nó **giải quyết một vấn đề thực tế**.

Hiểu được TẠI SAO quan trọng hơn hiểu được NHƯ THẾ NÀO. Vì khi bạn hiểu tại sao, bạn có thể tự tìm ra cách làm phù hợp với hoàn cảnh của mình.

> **"Code là công cụ. Tư duy là vũ khí thực sự."**

Tiếp theo: Phần 2: Tối Ưu & Phát Triển - "Nếu có thêm thời gian..."