Authorization Menggunakan Policy di Laravel

1. Memahami Policy Sebagai Konsep Otorisasi

1.1. Mengapa Policy Diperlukan?

Dalam aplikasi nyata, **tidak semua user yang sudah login boleh melakukan semua hal**. Login (authentication) hanya menjawab pertanyaan:

"Siapa dia?"

Namun aplikasi modern selalu membutuhkan jawaban lanjutan:

"Apa yang boleh dia lakukan?"

Di sinilah **otorisasi (authorization)** berperan. Laravel menyediakan beberapa mekanisme otorisasi, dan **Policy adalah mekanisme yang paling terstruktur dan paling tepat untuk aplikasi berbasis data (model-driven)**.

Tanpa Policy, developer pemula sering menulis logika seperti ini di controller:

```
if ($post->user_id !== auth()->id()) {
    abort(403);
}
```

Masalah pendekatan ini:

- Logika otorisasi tersebar di banyak controller
- Sulit dirawat ketika aturan berubah
- Controller menjadi gemuk dan tidak fokus
- Sulit dibaca dan diuji

Policy hadir untuk **mengangkat logika otorisasi keluar dari controller** dan menjadikannya **aturan formal yang terpusat**.

1.2. Apa Itu Policy Secara Konseptual?

Secara konseptual, **Policy adalah kumpulan aturan yang menjawab pertanyaan**:

"Apakah user ini boleh melakukan aksi tertentu terhadap data ini?"

Dengan kata lain, Policy adalah:

- **Penjaga akses (access guardian)** terhadap data
- **Mesin keputusan** yang mengembalikan *ya* atau *tidak*
- **Lapisan keamanan logis**, bukan visual dan bukan database

Policy **tidak**:

- menampilkan halaman
- menyimpan data
- memproses request
- memuat tampilan

Policy **hanya mengambil keputusan**.

1.3. Cara Berpikir Policy: Subjek, Aksi, dan Objek

Setiap Policy selalu bekerja dengan **tiga komponen konseptual utama**:

1. **Subjek (User)**
   Siapa yang sedang melakukan tindakan?
2. **Aksi (Action)**
   Tindakan apa yang ingin dilakukan?
   Contoh: `view`, `create`, `update`, `delete`
3. **Objek (Model)**
   Data apa yang menjadi sasaran tindakan?

Policy menjawab pertanyaan dalam bentuk:

"Apakah *User A* boleh melakukan *aksi X* terhadap *data Y*?"

Struktur berpikir inilah yang nanti diterjemahkan Laravel ke dalam bentuk method Policy.

1.4. Policy vs Pengecekan Manual di Controller

Perbandingan konseptual berikut penting untuk memperjelas posisi Policy.

Tanpa Policy (manual):

- Setiap controller membuat aturan sendiri
- Aturan mudah tidak konsisten
- Sulit dikembangkan untuk aplikasi besar

Dengan Policy:

- Aturan akses berada di satu tempat
- Controller hanya meminta keputusan
- Perubahan kebijakan tidak menyentuh controller

Controller tidak lagi *memutuskan*, tetapi **meminta izin**.

1.5. Policy Bukan Sekadar "If Statement"

Kesalahan umum yang biasa terjadi adalah menganggap Policy hanya sebagai:

```
if (...) return true;
else return false;
```

Padahal secara arsitektural, Policy:

- Mewakili **kebijakan aplikasi**
- Menjadi dokumentasi implisit tentang aturan akses
- Menjadi kontrak keamanan antara data dan user

Dalam tim pengembangan, Policy sering dibaca sebagai:

“Spesifikasi siapa boleh apa”

1.6. Kapan Policy Digunakan?

Policy paling tepat digunakan ketika**:**

- Akses bergantung pada **data tertentu**
- Hak akses berbeda untuk setiap record
- Aplikasi memiliki konsep kepemilikan data
- Aplikasi berkembang dan aturan akses bisa berubah

Contoh kasus nyata:

- User hanya boleh mengedit posting miliknya
- Admin boleh menghapus semua data
- User biasa hanya boleh melihat data tertentu

Semua kasus ini **tidak ideal jika ditangani langsung di controller**.

1.7. Posisi Policy dalam Alur Request Laravel

Secara konseptual, posisi Policy berada di antara:

- **Controller** (yang menerima request)
- **Model / Database** (yang menyimpan data)

Alurnya:

1. Request masuk
2. Controller dipanggil
3. Controller meminta izin ke Policy
4. Policy memberi keputusan
5. Jika diizinkan → proses dilanjutkan
6. Jika ditolak → request dihentikan

Dengan demikian, Policy adalah **gerbang logika sebelum data disentuh**.

2. Konsep Implementasi Policy

2.1. Apa yang Sebenarnya Dilakukan Policy?

Pada tahap ini, penting untuk mengubah cara pandang: **Policy bukan sekadar “pengecekan izin”**, melainkan **mekanisme pengambilan keputusan keamanan sebelum sebuah aksi bisnis dijalankan**.

Ketika di dalam controller kita menulis:

```
$this->authorize('update', $post);
```

baris ini tampak sederhana, tetapi di balik layar Laravel menjalankan **alur keputusan yang sistematis dan terstruktur**.

Secara konseptual, Laravel melakukan langkah-langkah berikut:

1. **Mengambil user yang sedang login**
   Laravel tidak meminta kita mengirimkan objek `User` secara eksplisit. Framework secara otomatis mengambil *authenticated user* dari sistem otentikasi (`Auth::user()`).
   Ini menunjukkan bahwa **Policy selalu berangkat dari konteks "siapa yang sedang bertindak"**.

2. **Mencari Policy yang terkait dengan model**
   Laravel kemudian melihat objek kedua yang dikirim (`$post`). Dari sini, Laravel:
   - Mengidentifikasi bahwa objek tersebut adalah instance dari model `Post`
   - Mencari apakah ada Policy yang terdaftar untuk model tersebut (misalnya `PostPolicy`)

   Jika tidak ditemukan, Laravel menganggap **tidak ada aturan yang mendefinisikan akses**, dan ini berpotensi menghasilkan error atau penolakan akses.

3. **Menjalankan method yang sesuai dengan aksi**
   Kata `'update'` bukan string sembarangan. Laravel memaknainya sebagai:
   "Cari method bernama `update` di dalam `PostPolicy`"
   Method tersebut dipanggil dengan **dua parameter penting**:
   - User yang sedang login
   - Data model yang akan diakses atau dimodifikasi

4. **Menggunakan nilai boolean sebagai keputusan akhir**
   Hasil dari method Policy **harus berupa boolean**:
   - `true` → aksi diizinkan
   - `false` → aksi ditolak (403 Forbidden)

   Tidak ada efek samping. Policy **tidak mengubah data**, **tidak menyimpan apa pun**, dan **tidak menampilkan tampilan**.

Hal yang sangat krusial untuk dipahami:

Policy selalu dijalankan sebelum data diubah, bukan sesudah.

Artinya:

- Jika Policy menolak → query `UPDATE`, `DELETE`, atau aksi sensitif lainnya **tidak pernah dijalankan**
- Policy bertindak sebagai *gerbang pengaman* (*security gate*) pada level logika aplikasi

2.2. Hubungan Policy dengan Model

Konsep berikut adalah **fondasi desain Policy** dan wajib dipahami dengan benar:

Satu Model memiliki satu Policy utama

Relasi ini bersifat **konseptual**, bukan teknis semata. Artinya:

- Model merepresentasikan **data**
- Policy merepresentasikan **aturan akses terhadap data tersebut**

Dengan demikian, Policy dapat dipandang sebagai:

“Buku peraturan tentang siapa boleh melakukan apa terhadap data ini.”

Contoh pemetaan yang lazim:

- Model `Post` → `PostPolicy`
  Mengatur:
  - siapa boleh melihat post
  - siapa boleh mengedit post
  - siapa boleh menghapus post
- Model `Product` → `ProductPolicy`
  Mengatur:
  - siapa boleh menambah produk
  - siapa boleh mengubah harga
  - siapa boleh menghapus produk

Dengan pendekatan ini:

- **Controller menjadi bersih**, karena tidak dipenuhi logika otorisasi
- **Aturan akses terpusat**, tidak tersebar di banyak file
- **Perubahan kebijakan** cukup dilakukan di satu tempat

Secara arsitektural, ini mencerminkan prinsip:

- *Separation of Concerns*
- *Single Responsibility Principle*

Model fokus pada data, Policy fokus pada **hak akses terhadap data**.

2.3. Struktur Method Policy

Setiap method di dalam Policy memiliki makna yang sangat spesifik dan konsisten.

a. Satu Method = Satu Aksi

Nama method **mewakili aksi yang ingin dilakukan**, misalnya:

```
- view
- create
- update
- delete
```

Laravel akan **mengasosiasikan nama aksi ini secara langsung** dengan pemanggilan `authorize()` atau `can()`.

Dengan demikian:

```
$this->authorize('update', $post);
```

selalu berarti:

“Apakah user ini boleh melakukan aksi *update* terhadap *post ini*?”

b. Parameter Method Policy

Struktur parameter hampir selalu sama:

```
public function update(User $user, Post $post)
```

Maknanya:

- `$user` → **aktor** (siapa yang bertindak)
- `$post` → **objek sasaran** (data apa yang akan dipengaruhi)

Ini mencerminkan pola berpikir keamanan:

Siapa melakukan apa terhadap data yang mana

c. Nilai Kembalian (Return Value)

Policy **harus mengembalikan boolean**:

- `true` → diizinkan
- `false` → ditolak

Tidak boleh:

- mengembalikan string
- melakukan redirect
- menampilkan view

Policy murni berfungsi sebagai **mesin keputusan**.

d. Pola Logika yang Paling Umum

Logika yang paling sering dijumpai adalah **kepemilikan data**:

```
return $user->id === $post->user_id;
```

Maknanya secara konseptual:

"User hanya boleh mengubah data yang dia miliki sendiri."

Pola ini sangat penting dalam aplikasi nyata seperti:

- blog
- sistem posting
- sistem transaksi
- manajemen data personal

Dengan satu baris logika ini:

- Controller tidak perlu tahu detail kepemilikan data
- Policy menjadi satu-satunya sumber kebenaran untuk aturan akses

3. Implementasi Aplikasi

Contoh ini akan membangun **aplikasi Post sederhana** di mana:

- User **harus login**
- User **hanya boleh mengedit dan menghapus Post miliknya sendiri**
- Aturan akses **tidak ditulis di controller**, tetapi **dipusatkan di Policy**

Framework yang digunakan adalah Laravel dengan database MySQL.

Bagian 1 — Membuat Project

3.1. Membuat Project Laravel Baru

Buka terminal, lalu jalankan:

```
composer create-project laravel/laravel laravel-policy-post
cd laravel-policy-post
```

Jalankan server untuk memastikan proyek berhasil dibuat:

```
php artisan serve
```

Jika halaman default Laravel tampil, proyek siap digunakan.

Bagian 2 — Konfigurasi Database

3.2. Membuat Database

Buat database baru di MySQL, misalnya:

```
CREATE DATABASE laravel_policy_post;
```

3.3. Konfigurasi File `.env`

Edit file `.env`:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_policy_post
DB_USERNAME=root
DB_PASSWORD=
```

Bagian 3 — Mengaktifkan Authentication

Mengaktifkan Authentication (Laravel Breeze)

Authorization **tidak mungkin ada tanpa Authentication**. Oleh karena itu, langkah pertama adalah mengaktifkan sistem login.

3.4. Install Breeze

```
composer require laravel/breeze --dev
php artisan breeze:install
npm install
npm run build
php artisan migrate
```

Hasil dari langkah ini:

- Tabel `users` dibuat
- Fitur login dan register aktif
- Middleware `auth` siap digunakan

Coba register satu atau dua user sebagai data uji.

Bagian 4 — Membuat Model Dan Migration Post

3.5. Membuat Model dan Migration

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

Edit file migration `database/migrations/...create_posts_table.php`:

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

Jalankan migration:

```
php artisan migrate
```

Makna penting:

- `user_id` menunjukkan **pemilik Post**
- Inilah dasar logika Policy nantinya

Bagian 5 — Relasi Antar Model

3.6. Model Post

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

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

3.7. Model User

Tambahkan di `User.php`:

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

Relasi ini sangat penting karena:

- Controller akan membuat Post lewat user
- Policy akan membandingkan `user_id`

Bagian 6 — Membuat Dan Memahami Policy

3.8. Membuat Policy

```
php artisan make:policy PostPolicy --model=Post
```

File akan dibuat di:

```
app/Policies/PostPolicy.php
```

3.9. Implementasi Detail `PostPolicy`

```php
class PostPolicy
{
    /**
     * Determine whether the user can view any models.
     */
    public function viewAny(User $user): bool
    {
        return true;
    }

    /**
     * Determine whether the user can view the model.
     */
    public function view(User $user, Post $post): bool
    {
        return true;
    }

    /**
     * Determine whether the user can create models.
     */
    public function create(User $user): bool
    {
        return true;
    }

    /**
     * Determine whether the user can update the model.
     */
    public function update(User $user, Post $post): bool
    {
        return $user->id === $post->user_id;
    }

    /**
     * Determine whether the user can delete the model.
     */
```

```
    public function delete(User $user, Post $post): bool
    {
        return $user->id === $post->user_id;
    }

}
```

3.10. Analisis Logika Policy (Bagian Paling Penting)

Penekanan konseptual:

1. **Policy tidak peduli UI**
   Policy tidak tahu apakah tombol “Edit” ada atau tidak.
2. **Policy tidak peduli Controller**
   Policy tidak tahu method mana yang memanggilnya.
3. **Policy hanya peduli aturan**
   “Apakah user ini boleh melakukan aksi ini terhadap data ini?”

Inilah yang membuat Policy:

- **Konsisten** → aturan sama di mana pun
- **Reusable** → dipakai di controller dan Blade
- **Aman** → tidak bisa dilewati hanya dengan manipulasi UI

Bagian 7 — Registrasi Policy (Laravel >= 11)

~~3.11. AuthServiceProvider (Laravel <= 10)~~

Ini untuk Laravel <= 10

~~Buat AuthServiceProvider:~~

```
php artisan make:provider AuthServiceProvider
```

~~Pastikan file `app/Providers/AuthServiceProvider.php` berisi:~~

```
protected $policies = [
    Post::class => PostPolicy::class,
];
```

~~Tanpa ini:~~

- ~~Laravel **tidak tahu** Post menggunakan PostPolicy~~

3.11. Auto-Discovery Policy (Default Laravel >= 11)

Pada Laravel >= 11, **Policy tidak perlu diregistrasi manual**.

Laravel akan otomatis menghubungkan:

- `Post` → `PostPolicy`

Selama:

- Policy berada di folder `app/Policies`

- Nama Policy sesuai konvensi

Tidak ada kode yang perlu ditulis di Provider apa pun.

Bagian 8 — Resource Controller

3.12. Membuat Controller

```
php artisan make:controller PostController --resource
```

3.13 Tambahkan Trait `AuthorizesRequests` ke Controller

Buka file `PostController.php`, lalu:

```
use Illuminate\Foundation\Auth\Access\AuthorizesRequests;

class PostController extends Controller
{
    use AuthorizesRequests;

    // method-method controller
}
```

3.14. Method `index`

```
use App\Models\Post;

public function index()
{
    $posts = Post::with('user')
                ->latest()
                ->get();

    return view('posts.index', compact('posts'));
}
```

3.15. Method `create`

```
public function create()
{
    $this->authorize('create', Post::class);

    return view('posts.create');
}
```

3.16. Method `store`

```
public function store(Request $request)
{
    $validated = $request->validate([
```

```
        'title'   => 'required',
        'content' => 'required',
    ]);

    $request->user()->posts()->create($validated);

    return redirect()->route('posts.index');
}
```

Catatan penting:

- `user_id` **tidak dikirim dari form**
- Data kepemilikan dikontrol oleh backend

3.17. Method `edit` (Authorization)

```
public function edit(Post $post)
{
    $this->authorize('update', $post);

    return view('posts.edit', compact('post'));
}
```

Artinya:

"Sebelum halaman edit ditampilkan, minta izin ke Policy."

3.18. Method `update`

```
public function update(Request $request, Post $post)
{
    $this->authorize('update', $post);

    $post->update($request->all());

    return redirect()->route('posts.index');
}
```

Jika Policy menolak:

- Update **tidak pernah dijalankan**
- Laravel otomatis mengembalikan **403 Forbidden**

3.19. Method `destroy`

```
public function destroy(Post $post)
{
    $this->authorize('delete', $post);

    $post->delete();

    return redirect()->route('posts.index');
}
```

Bagian 9 — Routing

3.20. Route Resource

```
Route::resource('posts', PostController::class)
    ->middleware('auth');
```

Makna:

- Semua fitur Post **wajib login**
- Authorization detail diatur oleh Policy

Bagian 10 — Blade Template

3.21. posts/plain.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Simple Blog</title>
    <style>
        body { font-family: Arial, sans-serif; }
        .container { width: 800px; margin: 40px auto; }

        label { display: block; margin-top: 10px; }
        input, textarea { width: 100%; padding: 6px; }
        button { margin-top: 15px; padding: 8px 14px; }

        table { width: 100%; border-collapse: collapse; }
        th, td { border: 1px solid #333; padding: 6px; }

        .user-bar {
            margin-bottom: 20px;
            padding: 10px;
            border: 1px solid #999;
            background: #f5f5f5;
        }

        .logout-form {
            display: inline;
        }
    </style>
</head>
<body>
    <div class="container">

        {{-- Informasi user login --}}
        @auth
            <div class="user-bar">
                Login sebagai:
                <strong>{{ auth()->user()->name }}</strong>
                ({{ auth()->user()->email }})

                <form class="logout-form" action="{{ route('logout') }}"
method="POST">
```

```
                    @csrf
                    <button type="submit">Logout</button>
                </form>
            </div>
        @endauth

        @guest
            <div class="user-bar">
                <em>Belum login</em>
            </div>
        @endguest

        {{-- Konten halaman --}}
        @yield('content')

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

3.21. posts/index.blade.php

```
@extends('layouts.plain')

@section('content')
<h2>Daftar Post</h2>

<a href="{{ route('posts.create') }}">Tambah Post</a>

<table>
    <tr>
        <th>Judul</th>
        <th>Aksi</th>
    </tr>

    @foreach ($posts as $post)
    <tr>
        <td>{{ $post->title }}</td>
        <td>
            <a href="{{ route('posts.edit', $post) }}">Edit</a>

            |

            <form action="{{ route('posts.destroy', $post) }}"
                  method="POST"
                  style="display:inline;">
                @csrf
                @method('DELETE')

                <button type="submit"
                        onclick="return confirm('Yakin ingin menghapus post
ini?')">
                    Hapus
                </button>
            </form>
        </td>
    </tr>
    @endforeach
```

```
</table>
@endsection
```

Fungsi

- Menampilkan daftar Post
- Menampilkan tombol **Edit / Hapus** berdasarkan Policy (`@can`)
- UX-only, **bukan keamanan utama**

3.22. posts/create.blade.php

```
@extends('layouts.plain')

@section('content')
<h2>Tambah Post</h2>

<form method="POST" action="{{ route('posts.store') }}">
    @csrf

    <label>Judul</label>
    <input type="text" name="title">

    <label>Konten</label>
    <textarea name="content"></textarea>

    <button type="submit">Simpan</button>
</form>
@endsection
```

Fungsi

- Menampilkan form tambah Post
- Tidak ada `user_id` di form (keamanan)
- Validasi ditampilkan jika gagal

3.23. posts/edit.blade.php

```
@extends('layouts.plain')

@section('content')
<h2>Edit Post</h2>

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

    <label>Judul</label>
    <input type="text" name="title" value="{{ $post->title }}">

    <label>Konten</label>
    <textarea name="body">{{ $post->body }}</textarea>

    <button type="submit">Update</button>
</form>
@endsection
```

Fungsi

- Menampilkan form edit Post
- Data lama otomatis terisi
- Akses **sudah dikunci oleh Policy di controller**

4. Alur Eksekusi & Pengujian

Bagian ini merupakan **titik kulminasi** dari seluruh pembahasan Policy. Pada tahap ini kita tidak lagi hanya "menulis kode", tetapi **memahami apa yang benar-benar terjadi ketika aplikasi dijalankan**, khususnya **bagaimana Laravel mengambil keputusan keamanan** terhadap sebuah aksi pengguna.

4.1 Alur Kerja Lengkap (Execution Flow)

Mari kita telusuri alur ini **sebagai sebuah cerita eksekusi sistem**, bukan sekadar daftar langkah.

1. User Login

Alur selalu dimulai dari **identitas pengguna**.

Ketika user berhasil login, Laravel menyimpan informasi user yang terautentikasi di dalam sistem (melalui session). Sejak saat itu:

```
auth()->user()
```

menjadi **sumber kebenaran identitas** di seluruh aplikasi.

Tanpa login, tidak ada konteks authorization. Authorization **selalu** berbasis *siapa* yang melakukan aksi.

2. User Membuka Halaman Edit Post

User kemudian mengakses URL seperti:

```
/posts/5/edit
```

Pada titik ini:

- Laravel melakukan *routing*
- Route memanggil `PostController@edit`
- Model `Post` dengan ID `5` berhasil diambil dari database

Namun, **penting** untuk dipahami:

Mengambil data ≠ boleh mengubah data

Laravel **belum mengambil keputusan keamanan** di sini.

3. Controller Memanggil `authorize()`

Di dalam method controller (misalnya `edit()` atau `update()`), terdapat baris:

```
$this->authorize('update', $post);
```

Inilah **titik kritis authorization**.

Dengan satu baris ini, developer **secara eksplisit menyerahkan keputusan keamanan ke Laravel**.

Controller **tidak tahu dan tidak peduli**:

- siapa pemilik post
- bagaimana logika pengecekan
- apa aturan bisnisnya

Controller hanya berkata:

"Laravel, pastikan user ini **berhak** melakukan aksi ini."

4. Laravel Memanggil `PostPolicy`

Laravel kemudian melakukan proses internal berikut:

1. Mengidentifikasi **model** yang terlibat (`Post`)
2. Menemukan policy yang sesuai (`PostPolicy`)
3. Mencocokkan **aksi** (`update`, `delete`, dll.)
4. Mengirimkan **dua objek penting** ke policy:
    - User yang sedang login
    - Post yang sedang diakses

Secara konseptual, Laravel bertanya:

"Apakah *User A* boleh melakukan *aksi X* terhadap *Post B*?"

5. Policy Mengecek Kepemilikan Data

Di dalam `PostPolicy`, logika biasanya sangat sederhana namun sangat bermakna:

```
return $user->id === $post->user_id;
```

Di sinilah **aturan bisnis authorization hidup**.

Maknanya:

- Authorization **tidak berbasis role semata**
- Tetapi berbasis **relasi data**
- User hanya boleh mengelola **data miliknya sendiri**

Inilah esensi *data ownership–based authorization*.

6. Jika Gagal → 403 Forbidden

Jika policy mengembalikan `false`, Laravel **secara otomatis**:

- Menghentikan eksekusi controller
- Tidak menjalankan view, update, atau delete
- Mengirim HTTP Response **403 Forbidden**

Penting ditekankan di sini:

> **403 bukan error aplikasi**
> 403 adalah **keputusan keamanan yang benar**

Ini berarti sistem **berfungsi dengan baik**.

7. Jika Berhasil → Aksi Dijalankan

Jika policy mengembalikan `true`:

- Eksekusi controller **dilanjutkan**
- Form edit ditampilkan
- Data disimpan
- Atau data dihapus

Dengan demikian:

- Controller tetap bersih
- Policy menjadi pusat logika keamanan
- Aplikasi konsisten dan aman

4.2 Skenario Uji Wajib (Authorization Testing)

Bagian ini menekankan bahwa **authorization harus diuji secara perilaku**, bukan hanya "berjalan tanpa error".

1. Edit Post Sendiri → Berhasil

Kondisi:

- User A login
- User A mengedit post miliknya sendiri

Hasil yang diharapkan:

- Halaman edit tampil
- Update berhasil disimpan

Makna akademik:

- Policy mengizinkan

- Relasi user–data benar
- Sistem bekerja sesuai desain

2. Edit Post User Lain → 403 Forbidden

Kondisi:

- User A login
- User A mencoba mengedit post milik User B

Hasil yang diharapkan:

- Akses ditolak
- Status 403 muncul

Makna penting:

- Tidak ada kebocoran hak akses
- Controller tidak “bocor logika”
- Policy benar-benar dieksekusi

3. Hapus Post Sendiri → Berhasil

Kondisi:

- User A menghapus post miliknya sendiri

Hasil:

- Data terhapus
- Redirect berhasil

Makna:

- Policy konsisten untuk aksi berbeda (delete)
- Authorization bersifat reusable

4. Hapus Post User Lain → 403 Forbidden

Kondisi:

- User A mencoba menghapus post milik User B

Hasil:

- Ditolak oleh sistem

Makna konseptual:

- Policy bukan hanya formalitas
- Sistem benar-benar menjaga integritas data

Penekanan

1. **Authorization bukan UI**
    - Tombol boleh disembunyikan
    - Tapi keputusan final **selalu di server**
2. **Policy = penjaga data**
    - Bukan sekadar fitur tambahan
    - Tapi komponen inti keamanan aplikasi
3. **403 adalah keberhasilan desain**
    - Menandakan sistem tegas dan konsisten

5. Penegasan Akhir

1. Policy Wajib untuk CRUD Berbasis Data

Setiap operasi CRUD yang:

- melibatkan **data milik user**
- memungkinkan **user mengubah atau menghapus data**

WAJIB memiliki mekanisme Policy.

Tanpa Policy:

- siapa pun yang lolos autentikasi berpotensi mengakses data yang bukan miliknya
- aplikasi rentan terhadap *horizontal privilege escalation*

Kesimpulan:
CRUD tanpa Policy pada aplikasi multi-user adalah **cacat desain**, bukan sekadar kekurangan fitur.

2. Jangan Menaruh Logika Authorization di Controller

Controller **bukan tempat pengambilan keputusan keamanan**.

Jika Anda menulis:

```
if ($post->user_id !== auth()->id()) {
    abort(403);
}
```

maka:

- logika tersebar
- sulit dirawat
- rawan lupa diterapkan di method lain
- melanggar prinsip *single responsibility*

Controller seharusnya hanya:

- menerima request

- memanggil authorization
- menjalankan aksi jika diizinkan

Controller bertanya, Policy memutuskan.

3. Jangan Mengandalkan Blade sebagai Keamanan

Blade directive seperti:

```
@can('update', $post)
```

bukan mekanisme keamanan utama.

Blade hanya:

- mengatur tampilan
- meningkatkan UX
- mencegah kebingungan user

Namun:

- request bisa dipalsukan
- URL bisa diakses langsung
- form bisa dikirim manual

Jika tidak ada authorize() di controller, aplikasi tetap tidak aman.

Blade **melengkapi**, bukan **menggantikan** Policy.

4. Policy adalah Praktik Profesional Laravel

Menggunakan Policy berarti Anda:

- mengikuti arsitektur resmi Laravel
- menulis kode yang:
    - konsisten
    - terstruktur
    - mudah diuji
    - mudah dikembangkan
- siap bekerja pada:
    - tim besar
    - proyek jangka panjang
    - aplikasi skala produksi

Di dunia kerja:

- reviewer akan mencari Policy
- codebase tanpa Policy dianggap *amateur-level*
- Policy adalah *baseline competency*, bukan keunggulan tambahan

Policy bukan fitur opsional. Policy adalah standar profesional.

Daftar Pustaka

1. Laravel.
   Authorization — Policies.
   Dokumentasi resmi Laravel.
   Tersedia di:
   https://laravel.com/docs/authorization
2. Laravel.
   Authentication Starter Kits (Breeze).
   Dokumentasi resmi Laravel.
   Tersedia di:
   https://laravel.com/docs/starter-kits
3. Laravel.
   Controllers & Authorization Helpers.
   Dokumentasi resmi Laravel.
   Tersedia di:
   https://laravel.com/docs/controllers