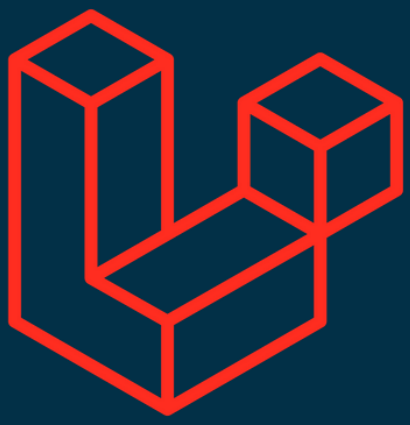

Belajar Membuat Aplikasi Laravel Sesuai Standar Best Practice

Asdita Prasetya - @codingtengahmalam

Table of contents

Pembukaan

Sebagai developer Laravel, seringkali kita menghadapi beberapa masalah seperti: kode yang sulit dibaca, performa aplikasi yang menurun seiring pertumbuhan pengguna, struktur kode yang makin rumit, dan masih banyak lagi.

Masalah ini seringkali disebabkan oleh kurangnya pemahaman terhadap best practice dalam pengembangan aplikasi Laravel. Maka dari itu saya menulis buku dengan judul **"Belajar Membuat Aplikasi Laravel Sesuai Standar Best Practice"** Harapan saya buku ini hadir untuk membantu mengatasi masalah yang muncul dalam pengembangan aplikasi, selain itu juga membantu developer Laravel berkembang lebih baik lagi dengan tidak hanya fokus pada pengembangan aplikasi saja, tetapi juga memperhatikan best practice dalam pengembangan aplikasi.

Buku ini berisi berbagai best practices yang bisa langsung diterapkan, mulai dari tips pengelolaan kode, desain arsitektur, hingga teknik optimalisasi performa dan keamanan

Ebook ini cocok untuk semua tingkatan developer Laravel: dari pemula yang ingin menerapkan standar yang baik, hingga developer profesional yang ingin menyempurnakan aplikasi berskala besar.

Sebagai bonus, ebook ini dilengkapi dengan studi kasus proyek nyata: aplikasi donasi online yang terintegrasi dengan pembayaran otomatis Midtrans. Anda akan dibimbing melalui setiap langkah, memahami praktik terbaik dalam proyek nyata yang siap produksi.

Dengan panduan ini, Anda bisa meningkatkan keterampilan Laravel dan memastikan aplikasi yang Anda bangun lebih profesional, aman, dan scalable!

Hak Cipta & Ketentuan Penggunaan

Aturan Penamaan Laravel

Laravel adalah framework yang telah di desain sedermikian rupa untuk membuat developer menjadi lebih produktif, efektif dan membuat kita hanya fokus untuk mengembangkan produk saja.

Untuk meningkatkan produktivitas hal yang harus dilakukan adalah menulis kode sesuai dengan standar Laravel. baik bekerja secara tim maupun individu, mengikuti standar kode merupakan investasi.

Mungkin sebagian developer yang belum terbiasa akan merasa mengikuti standar malah mempersulit proses development. Namun percayalah, ketika project kamu sudah selesai dan suatu hari nanti kamu perlu membuka-nya kembali jika kamu tidak mengikuti standar yang ada, kamu akan membuang banyak waktu.

Selain itu, mengikuti konvensi penamaan Laravel juga meningkatkan keamanan dan performa aplikasi. Dengan struktur yang konsisten, lebih mudah untuk mendeteksi dan memperbaiki potensi masalah keamanan. Performa juga dapat ditingkatkan karena kode yang terorganisir dengan baik lebih mudah dioptimalkan dan di-cache.

Saat ini Laravel menerapkan **PSR-12: Extended Coding Style** standar ini berisikan ketentuan yang sangat mendetail mengenai gaya untuk menulis kode PHP.

Berikut ini adalah contoh konversi penamaan yang sesuai dengan standar Laravel

Bahasa

Gunakan **bahasa Inggris** untuk semua komponen dalam Laravel.

Database - Nama Tabel: Gunakan kata jamak (contoh: `donations`).
- Nama Kolom: Gunakan snake_case (contoh: `short_description`).
- Primary Key: Gunakan `id` sebagai primary key secara default.

Model & Controller - Model: Gunakan kata tunggal (contoh: `Donation`).
- Controller: Gunakan kata tunggal dengan akhiran `Controller` (contoh: `DonationController`).

Method, Variable, dan Route - Method: Gunakan camelCase (contoh: `donationPackages`).
- Variabel: Gunakan camelCase (contoh: `$currentDonation`).
- Route:
- Gunakan kata jamak (contoh: `Route::resource('donations')`).
- Gunakan snake_case dengan notasi dot untuk nama router (contoh: `donations.show_package`).

Relationship dalam Eloquent - hasOne, belongsTo: Gunakan kata tunggal (contoh: `author`).
- hasMany, belongsToMany: Gunakan kata jamak (contoh: `donations`, `payments`).

Seeder & View - Seeder Gunakan kata tunggal dengan akhiran `Seeder` (contoh: `DonationSeeder`).
- View: Gunakan kebab-case untuk penamaan file (contoh: `show-detail-donation.blade.php`).

Resource Controller - Gunakan `Route::resource('donations', DonationController::class)` hanya jika controller digunakan untuk mengelola resource.

Dan masih banyak lagi konvensi penamaan yang perlu diperhatikan dalam Laravel.

Dengan mengikuti konvensi penamaan yang konsisten seperti yang dijelaskan di atas, projectmu akan menjadi lebih terstruktur dan mudah dipahami. Hal ini tidak hanya membantu dalam pemeliharaan kode jangka panjang, tetapi juga memudahkan kolaborasi dengan developer lain.

Konsep DRY (Don't Repeat Yourself)

Konsep **DRY (Don't Repeat Yourself)** dalam pemrograman berarti menghindari pengulangan kode atau logika yang sama di berbagai tempat. Tujuannya adalah membuat kode lebih efisien, mudah dipelihara, dan bebas dari bug.

Dengan menerapkan prinsip DRY, kita bisa mengubah atau memperbaiki logika hanya pada satu tempat saja, tanpa harus mencari dan mengubah kode yang sama di banyak lokasi.

Penerapan DRY dalam Kehidupan Sehari-hari

Konsep DRY juga bisa diterapkan dalam kehidupan sehari-hari untuk meningkatkan efisiensi dan produktivitas. Salah satu contoh sederhana adalah dalam membuat daftar belanja mingguan. Mari kita lihat bagaimana konsep DRY bisa membantu mengorganisir kegiatan rutin dengan lebih efektif.

Daripada menulis ulang daftar belanja setiap minggu dari awal, kita bisa menyimpan satu daftar belanja mingguan yang berisi barang-barang kebutuhan rutin, seperti beras, telur, susu, dan sayuran. Setiap minggu, cukup buka daftar tersebut dan tambahkan atau kurangi barang sesuai kebutuhan saat itu.

```
 Daftar Belanja Mingguan

Kebutuhan Pokok
- [ ] Beras - 5 kg
- [ ] Telur - 1 lusin
- [ ] Susu UHT - 3 kotak
- [ ] Minyak goreng - 2 liter
- [ ] Gula pasir - 1 kg

Sayuran & Buah
- [ ] Wortel - 500 gr
- [ ] Bayam - 1 ikat
- [ ] Tomat - 500 gr
- [ ] Pisang - 1 sisir
```

```
Protein
- [ ] Daging ayam - 1 kg
- [ ] Tahu - 10 potong
- [ ] Tempe - 2 papan

Kebutuhan Rumah Tangga
- [ ] Sabun cuci piring
- [ ] Deterjen
- [ ] Pasta gigi
```

Keuntungan: KIta tidak perlu mengulang penulisan barang yang sama berulang kali, sehingga menghemat waktu dan memastikan tidak ada barang yang terlewat.

Penerapan DRY dalam Aplikasi

Dalam aplikasi penggalangan dana, kita sering membutuhkan data total donasi yang ditampilkan di berbagai tempat seperti Dashboard, Report, detail donasi, dan riwayat donasi pengguna. Berikut adalah contoh penerapan dengan dan tanpa konsep DRY:

Tanpa menerapkan konsep DRY

```php
// DashboardController
$totalDonations = Donation::sum('amount');

// ReportController
$totalDonations = Donation::sum('amount');

// UserController
$totalDonations = Donation::where('user_id', $userId)->sum('amount');
```

Dengan menerapkan konsep DRY

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Donation extends Model
{
    // Reusable method
    public static function getTotalDonations($userId = null)
```

```
    {
        $query = self::query();

        if ($userId) {
            $query->where('user_id', $userId);
        }

        return $query->sum('amount');
    }
}
```

Maka pada tiap - tiap controller kita cukup memanggil method yang sudah kita buat sebelumnya, seperti contoh dibawah ini:

```
// DashboardController
$totalDonations = Donation::getTotalDonations();

// ReportController
$totalDonations = Donation::getTotalDonations();

// UserController
$totalDonations = Donation::getTotalDonations($userId);
```

Keuntungan

Dengan menerapkan konsep DRY, kita mendapatkan beberapa keuntungan: 1. Mudah Dikelola: Perubahan logika hanya perlu dilakukan di satu tempat. 2. Konsisten: Seluruh bagian aplikasi menggunakan logika yang seragam. 3. Efisien: Menghilangkan pengulangan kode yang tidak perlu.

Penerapan DRY menjadikan aplikasi lebih modular dan mudah dikembangkan di masa depan.

Selain itu, DRY juga mendorong pengembang untuk berpikir lebih sistematis dalam merancang solusi, sehingga menghasilkan arsitektur aplikasi yang lebih baik. Dengan menerapkan prinsip ini secara konsisten, tim pengembang dapat fokus pada fitur-fitur baru daripada menghabiskan waktu untuk memperbaiki kode yang berulang.

Implementasi Gaya Koding SOLID

Salah satu konsep yang sering dipakai dalam pengembangan perangkat lunak adalah SOLID. Konsep ini pertama kali diperkenalkan oleh Robert C. Martin pada tahun 2000-an lewat bukunya, "Design Principles and Design Patterns.". Berikut ini adalah keuntungan jika kita menggunakan konsep SOLID dalam pengembangan perangkat lunak:

1. Kode lebih mudah dipahami dan dirawat
2. Menghindari conflict saat bekerja dengan team
3. Perubahan kode menjadi lebih aman
4. Kode lebih mudah untuk di test dengan menggunakan unit test

SOLID terdiri dari 4 konsep utama, yaitu Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), dan Interface Segregation Principle (ISP). Mari kita bahas satu per satu konsep tersebut.

Single responsibility principle

Setiap class atau fungsi pada kode wajib memiliki satu tanggung jawab dan hanya satu alasan untuk diubah. Tujuannya adalah meningkatkan kohesi, mengurangi kompleksitas, dan memudahkan modifikasi. Dengan kata lain, suatu fungsi **tidak diperbolehkan** mengerjakan banyak hal,

Sebagai contoh, fungsi yang mendefinisikan struktur data hanya akan diubah saat ada perubahan pada struktur data tersebut, sedangkan *class/fungsi* yang menangani *business logic* hanya akan dimodifikasi ketika terjadi perubahan pada *business logic*.

Di Laravel, kita mengenal Form Request yang, jika diimplementasikan dengan tepat, akan sesuai dengan konsep SRP. Berikut adalah contoh penerapannya:

```
// form request

class DonationRequest extends FormRequest
{
    public function rules(): array
    {
        return [
```

```
            'campaign_id' => 'required',
            'amount' => 'required|numeric',
            'message' => 'nullable|string',
            'is_visible' => 'boolean'
        ];
    }
}

// Controller
namespace App\Http\Controllers;

class CampaignController extends Controller
{
    public function donateCampaign(DonationRequest $request)
    {
        $validatedData = $request->validated()
        $campaign = Campaign::findOrFail($request->campaign_id);

        // logic lainnya
    }
}
```

Pada contoh di atas, proses validasi data dipisahkan dari controller dan ditempatkan pada class terpisah yaitu `DonationRequest`. Dengan demikian, controller dapat fokus pada penanganan logika bisnis utama, sedangkan validasi data dikelola secara independen. Pemisahan ini menghasilkan kode yang lebih terorganisir dan mudah dirawat.

Dengan menerapkan SRP, kita juga mendapatkan keuntungan dalam hal pengujian, karena setiap komponen dapat diuji secara terpisah dan lebih mudah untuk memastikan bahwa setiap bagian berfungsi sesuai dengan yang diharapkan. Selain itu, ketika ada kebutuhan untuk mengubah logika validasi, kita hanya perlu fokus pada `DonationRequest` tanpa khawatir akan mempengaruhi logika bisnis di controller.

Open/closed principle

Software entities (kelas, modul, fungsi, dll.) sebaiknya terbuka untuk ekstensi tetapi tertutup untuk modifikasi, kita dapat menambahkan fitur baru tanpa harus mengubah kode yang sudah ada.

Hal ini membantu kita menghindari bug yang mungkin muncul akibat modifikasi kode yang telah berjalan. Memodifikasi kode yang sudah ada berisiko menimbulkan *bug* di *production*. Selain itu, kita harus melakukan *testing* ulang pada kode lama—sesuatu yang seharusnya tidak

perlu jika menerapkan *Open-closed Principle*. Prinsip ini mendorong kita untuk menulis kode yang modular.

Contoh kasus penerapan konsep OCP, perhatikan kode dibawah ini

```
public function donateCampaign(DonationRequest $request)
{
    $validatedData = $request->validated()
    $campaign = Campaign::findOrFail($request->campaign_id);

    $donation = Donation::create([
        'campaign_id' => $campaign->id,
        'user_id' => auth()->id(),
        'amount' => $request->amount,
        'message' => $request->message,
        'is_visible' => $request->is_visible,
        'campaign_detail' => json_encode($campaign),
        'status' => Donation::STATUS_PENDING,
    ]);

    $midtransSnapClient = new MidtransSnap($donation);
    $snapLink = $midtransSnapClient->getSnapLink();

    $donation->update([
        'payment_link' => $snapLink->redirect_url,
        'payment_detail' => json_encode($snapLink)
    ]);

    return response()->json([
        'redirect_url' => $snapLink->redirect_url
    ]);
}
```

Kode di atas menangani donasi untuk campaign dengan pembayaran melalui **Midtrans**. Namun, kode ini **tidak mengikuti Open/Closed Principle (OCP)** karena hanya mendukung satu metode pembayaran. Ketika kita ingin menambahkan metode pembayaran baru seperti **PayPal** atau **Manual** Payment, kita terpaksa **mengubah kode yang sudah ada**.

Penerapan OCP

Agar lebih fleksibel, dan mengikuti konsep OCP, kita bisa memisahkan logika pembayaran menggunakan Polymorphism dan Dependency Injection.

Buat Interface untuk Pembayaran

```php
interface PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation);
}
```

Implementasi sesuai dengan metode pembayaran

```php
// Midtrans payment
class MidtransGateway implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
        $midtransSnapClient = new MidtransSnap($donation);
        return $midtransSnapClient->getSnapLink()->redirect_url;
    }
}

// Paypal payment
class PaypalGateway implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
       // integrasi paypay
       return "https://paypal.com/checkout?...."
    }
}

// handle manual payment
class ManualPaymentGateway implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
       // integrasi paypay
       return "https://beramal.com/payment?...."
    }
}
```

Ubah fungsi `donateCampaign` untuk mendukung multiple payment gateway

```php
public function donateCampaign(DonationRequest $request)
{
    $validatedData = $request->validated()

    $campaign = Campaign::findOrFail($request->campaign_id);

    $donation = Donation::create([
        'campaign_id' => $campaign->id,
        'user_id' => auth()->id(),
```

```
        'amount' => $request->amount,
        'message' => $request->message,
        'is_visible' => $request->is_visible,
        'campaign_detail' => json_encode($campaign),
        'status' => Donation::STATUS_PENDING,
    ]);

    // Pilih gateway pembayaran berdasarkan request
    $paymentMethod = match ($request->payment_method) {
        'midtrans' => new MidtransGateway(),
        'paypal' => new PaypalGateway(),
        'manual' => new ManualPaymentGateway (),
        default => throw new \Exception("Metode pembayaran tidak didukung")
    };

    // Dapatkan link pembayaran dari gateway yang dipilih
    $paymentLink = $paymentMethod->getPaymentLink($donation);

    $donation->update([
        'payment_link' => $paymentLink,
        'payment_detail' => json_encode(['redirect_url' => $paymentLink ])
    ]);

    return response()->json([
        'redirect_url' => $paymentLink
    ]);
}
```

Dengan implementasi ini, fungsi `donateCampaign` telah menerapkan konsep Open/closed principle dengan baik. Ketika ada kebutuhan menambah payment gateway baru di masa depan, kita hanya perlu membuat class baru yang mengimplementasikan interface yang sudah ada dan menambahkan payment method-nya.

Liskov substitution principle (LSP)

Konsep ini berkaitan dengan *inheritance* dalam pemrograman. LSP menyatakan bahwa ketika suatu class mewarisi class induk (baik interface maupun class lainnya), class turunan tersebut harus dapat berfungsi dengan benar tanpa mengubah perilaku yang sudah ada. Mari kita lihat contoh implementasi LSP berikut.

Sebagai contoh, kita memiliki interface `PaymentGatewayInterface` yang telah kita buat sebelumnya. Setiap class yang mengimplementasikan interface ini (MidtransGateway, PaypalGateway, dan ManualPaymentGateway) dapat saling menggantikan tanpa mengganggu fungsionalitas sistem.

Namun kemudian kita akan menambahkan metode "Donasi Gratis", dimana donasi tidak memerlukan pembayaran sehingga tidak mengembalikan link pembayaran, contohnya seperti dibawah ini

```
class FreeDonationPayment implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
        // Tidak ada pembayaran, langsung set status sukses
        $donation->update(['status' => Donation::STATUS_SUCCESS]);
        return null; // Tidak ada URL pembayaran!
    }
}
```

Kode di atas berpotensi merusak alur pembayaran donasi karena `FreeDonationPayment` mengembalikan nilai null

Penerapan LSP

Ketika melakukan pembayaran dengan metode free payment user tidak membutuhkan link unuk menuju ke halaman pembayaran. Karena metode pembayarannya yang berbeda maka tidak bisa untuk menggunakan `PaymentGatewayInterface` , solusinya adalah dengan membuat interface baru, sehingga kita memiliki interface

1. Pembayaran Online `PaymentGatewayInterface`
2. Pembayaran Gratis `DirectConfirmationPaymentInterface`

Kurang lebih isi dari interface-nya sebagai berikut

```
// interface
interface PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation);
}

interface DirectConfirmationPaymentInterface {
    public function confirmPayment(Donation $donation);
}

// buat class sesuai dengan interface
class MidtransGateway implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
```

```
        return "https://midtrans.com/pay/{$donation->id}";
    }
}

class PayPalGateway implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
        return "https://paypal.com/checkout?donation_id={$donation->id}";
    }
}

class FreeDonationPayment implements DirectConfirmationPaymentInterface {
    public function confirmPayment(Donation $donation) {
        $donation->update(['status' => Donation::STATUS_SUCCESS]);
        return "Donasi berhasil dikonfirmasi tanpa pembayaran.";
    }
}
```

Maka kita akan menyesuaikan fungsi `donateCampaign` menjadi seperti dibawah ini

```
$paymentMethods = [
    'midtrans' => new MidtransPayment(),
    'paypal' => new PayPalPayment(),
    'free_donation' => new FreeDonationPayment(),
];

if (!isset($paymentMethods[$request->payment_method])) {
    throw new \Exception("Metode pembayaran tidak didukung");
}

$paymentMethod = $paymentMethods[$request->payment_method];

if ($paymentMethod instanceof PaymentGatewayInterface) {
    $snapLink = $paymentMethod->getPaymentLink($donation);
    $donation->update(['payment_link' => $snapLink]);

    return response()->json(['redirect_url' => $snapLink]);
}

if ($paymentMethod instanceof DirectConfirmationPaymentInterface) {
    $message = $paymentMethod->confirmPayment($donation);
    return response()->json(['message' => $message]);
}
```

```
// ...
```

Dengen begini tidak ada error saat menganti metode pembayaran karena `FreeDonation` tidak lagi dipaksa menggunakan sistem yang salah, selain itu kita bisa menambah metode pembayaran lain tanpa mengubah kode utama.

Interface segregation principle

Dalam bahasa inggris *segregation* memiliki arti *keeping things separated.* jika dikaitkan dengan *Interface segregation principle* memiliki arti pemisahan *interface*.

Dalam pembuatan *interface* lebih baik membuat banyak interface dengan fungsi yang spesifik, hal ini lebih baik daripada membuat satu *interface* dengan fungsi yang tidak spesifik. Tujuan dari pemisahan *interface* adalah untuk tidak memaksa *client* menggunakan kode yang tidak diperlukan.

Misalnya, kita punya satu interface besar untuk semua jenis pembayaran:

```
interface PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation);
    public function refund(Donation $donation);
}

// implementasi

class MidtransPayment implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
        return "https://midtrans.com/pay/{$donation->id}";
    }

    public function refund(Donation $donation) {
        return "Refund diproses melalui Midtrans";
    }
}

class FreeDonationPayment implements PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation) {
        // Tidak butuh link pembayaran
        return null;
    }
```

```
    public function refund(Donation $donation) {
        // Tidak mungkin ada refund untuk donasi gratis
    }
}
```

Pada `FreeDonationPayment` dipaksa untuk menggunakan `PaymentGatewayInterface` dan mengimplemenrasikan metode `refund` padahal tidak ada refund untuk donasi gratis, hal ini melanggar Interface Segregation Principle karena ada fungsi yang tidak diperlukan oleh semua class dalam interface

Lalu solusi yang benar adalah dengan memisahkan interface menjadi lebih kecil, seperti berikut ini

```
interface PaymentGatewayInterface {
    public function getPaymentLink(Donation $donation);
}

interface RefundablePaymentInterface {
    public function refund(Donation $donation);
}

interface DirectConfirmationPaymentInterface {
    public function confirmPayment(Donation $donation);
}

// lalu dalam implementasinya

class MidtransPayment implements PaymentGatewayInterface, RefundablePaymentInterface {
    public function getPaymentLink(Donation $donation) {
        return "https://midtrans.com/pay/{$donation->id}";
    }

    public function refund(Donation $donation) {
        return "Refund diproses melalui Midtrans untuk donasi ID: {$donation->id}";
    }
}

class PayPalPayment implements PaymentGatewayInterface, RefundablePaymentInterface {
    public function getPaymentLink(Donation $donation) {
        return "https://paypal.com/checkout?donation_id={$donation->id}";
    }
```

```
    public function refund(Donation $donation) {
        return "Refund diproses melalui PayPal untuk donasi ID: {$donation->id}";
    }
}

class FreeDonationPayment implements DirectConfirmationPaymentInterface {
    public function confirmPayment(Donation $donation) {
        $donation->update(['status' => Donation::STATUS_SUCCESS]);
        return "Donasi berhasil dikonfirmasi tanpa pembayaran.";
    }
}
```

Sekarang kita telah menerapkan Interface Segregation Principle (ISP) dengan baik, dengan detail sebagai berikut:

1. `MidtransPayment` dan `PayPalPayment` hanya mengimplementasikan metode yang dibutuhkan.
2. `FreeDonationPayment` tidak perlu mengimplementasikan fungsi `getPaymentLink()` dan `refund()` yang tidak relevan.
3. Kode menjadi lebih bersih, modular, dan mudah dikembangkan.
4. Penambahan metode pembayaran baru bisa dilakukan tanpa mengubah kode utama.

Mutator dan Accessor untuk Format Tanggal

Seringkali kita menyimpan data waktu di dalam database, misal kita punya tabel donations, akan ada kolom `payment_at, confirmed_at` dan yang lainnya. baiknya dalam menyimpan data waktu kita hanya perlu untuk menyimpan data tersebut sesuai dengan standar waktu yang ada dalam laravel.

Standar waktu ini biasanya menggunakan format `ISO 8601 (YYYY-MM-DD HH:MM:SS)`. Dengan menyimpan data dalam format standar, kita dapat dengan mudah memanipulasi dan menampilkan tanggal sesuai kebutuhan menggunakan fitur Accessors dan Mutators di Laravel. Ini memungkinkan kita untuk mengubah format tanggal saat mengambil atau menyimpan data tanpa mengubah struktur database.

Berikut adalah contoh penggunaan Accessors dan Mutators untuk mengelola format tanggal:

```
// Model
protected $casts = [
    'payment_at' => 'datetime',
];

/**
* Accessor: Formatkan kolom payment_at saat membaca
* (jika ingin dibaca dalam ISO 8601)
*/
public function getPaymentAtAttribute($value)
{
    return Carbon::parse($value)->toIso8601String();
}

/**
* Mutator: Ubah nilai payment_at menjadi format ISO 8601 sebelum disimpan
*/
public function setPaymentAtAttribute($value)
{
    $this->attributes['payment_at'] = Carbon::parse($value)->toIso8601String();
}

// View
```

```
{{ $object->payment_at->toDateString() }}
{{ $object->payment_at }}
```

Dengan menggunakan Accessors dan Mutators, kita dapat dengan mudah memanipulasi format tanggal tanpa mengubah data asli di database. Ini memberikan fleksibilitas dalam menampilkan tanggal sesuai kebutuhan aplikasi, seperti format yang berbeda untuk tampilan user dan format standar untuk penyimpanan atau perhitungan.

Menggunakan Mass Assignment

Mass Assignment memungkinkan kita mengisi beberapa atribut model sekaligus dalam satu langkah dengan menggunakan array.

Metode ini jauh lebih efisien daripada mengisi atribut satu per satu.

Dengan Mass Assignment, kita dapat mengirim array data ke model lalu Laravel akan secara otomatis menetapkan nilai-nilai tersebut ke atribut yang sesuai. Ini sangat berguna ketika bekerja dengan form input yang memiliki banyak field. Namun, penting untuk berhati-hati dan menggunakan fitur ini dengan bijak untuk menghindari masalah keamanan.

Untuk menggunakan Mass Assignment dengan aman, Laravel menyediakan `$fillable` dan `$guarded` pada model.

1. `$fillable` mendefinisikan atribut yang diizinkan untuk diisi secara massal,
2. `$guarded` mendefinisikan atribut yang tidak boleh diisi secara massal.

Dengan mengonfigurasi salah satu dari keduanya, kita dapat mencegah pengisian atribut yang tidak diinginkan dan meningkatkan keamanan aplikasi. Selain itu, selalu validasi input pengguna sebelum menggunakannya dalam Mass Assignment untuk menambah lapisan keamanan.

Berikut ini adalah contoh dalam menggunakan dan tanpa menggunakan

```
// Definisikan pada model
protected $fillable = ['name', 'email', 'password'];

// Dengan Mass Assignment
User::create([
    'name' => 'John Doe',
    'email' => 'john@example.com',
    'password' => bcrypt('secret'),
]);

// Tanpa Mass Assignment
$user = new User();
$user->name = 'John Doe';
$user->email = 'john@example.com';
```

```
$user->password = bcrypt('secret');
$user->save();
```

Selain itu kita juga bisa melakukan mengkombinasikan-nya dengan mengembalikan hasil dari validasi request, seperti contoh dibawah ini

```
$validatedData = $request->validate([
   'name' => 'required|string|max:255',
   'email' => 'required|email|unique:users,email',
   'password' => 'required|string|min:8',
    ]);

User::create([
    ...$validatedData,
    'password' => bcrypt($validatedData['password']),
]);
```

Dengan menggunakan Mass Assignment yang dikombinasikan dengan validasi request, kita dapat memastikan data yang masuk sudah tervalidasi sebelum disimpan ke database. Hal ini tidak hanya membuat kode lebih ringkas, tetapi juga meningkatkan

keamanan dan maintainability aplikasi. Pendekatan ini sangat direkomendasikan untuk menangani input form dalam aplikasi Laravel.

Hindari Penggunaan Magic String

Magic string adalah penulisan nilai string secara langsung dalam kode program tanpa mendefinisikannya di satu tempat terpusat.

Praktik ini merupakan pendekatan yang tidak disarankan dalam pengembangan perangkat lunak karena dapat menyebabkan berbagai masalah dalam pemeliharaan dan pengembangan kode. Ketika nilai string ditulis secara langsung di berbagai tempat dalam kode, hal ini dapat membuat kode menjadi sulit untuk dikelola dan rentan terhadap kesalahan. Contoh magic string sebagai berikut:

```
class User
{
    private string $type;

    public function __construct(string $type)
    {
        $this->type = $type;
    }

    public function isNormal(): bool
    {
        return $this->type === 'normal'; // Magic string
    }
}
```

Kekurangan Magic String

Penggunaan magic string baiknya dihindari, karena memiliki banyak kekurangan, diantaranya adalah sebagai berikut:

Rentan terhadap typo

String yang di-hardcode tidak memiliki validasi dari compiler atau IDE. Jika typo kesalahan ini hanya akan terdeteksi saat runtime, yang bisa menyebabkan bug yang sulit dilacak.

Sulit di pelihara (Maintainability)

Jika string digunakan pada banyak tempat, maka ketika ada perubahan pada string tersebut memerluklan perubahan juga pada semua tempat yang menggunakannya, dengan begitu dapat meningkatkan resiko kesalahan dan membutuhkan waktu untuk maintain hal tesebut.

Contohnya terdapat validasi pada **variable `$status`**, yang sebelumnya adalah `'normal'` menjadi `'regular'` maka harus mencari dan menggubah semua string `"normal"` menjadi `"regular"`

Sulit dipahami tool Editor

Dengan menggunakan magic string, Tool Editor code seperti PHPStrom, VScode, dan lainnya sulti untuk memahami project yang sedang dibuka, hal ini menjadikan tidak bisa menjalankan feature refactoring, automcomplate dan pengecekan tipe tipe data

Konteks tidak Jelas

String yang ditulis hardcode tidak memberkan konteks yang jelas, pembaca kode harus berfikir dan menebak apa maksud dari string tersebut.

Contohnya jika kita diberikan “nomal”,”regular”,”ordinary” atau “expected” tanpa ada konteks yang jelas hal ini sangat membingungkan

Cara menghindari Magic String

Untuk menghindari penggunaan magic string, kita dapat menggunakan beberapa pendekatan berikut:

Menggunakan Konstanta

Definisikan string yang sering digunakan sebagai konstanta dalam class:

```php
class User
{
    private const TYPE_NORMAL = 'normal';
    private string $type;

    public function __construct(string $type)
    {
        $this->type = $type;
    }

    public function isNormal(): bool
    {
        return $this->type === self::TYPE_NORMAL; //
```

```
    }
}
```

Menggunakan Enum

Gunakan enum untuk membatasi nilai yang valid:

```
enum UserType: string
{
    case NORMAL = 'normal';
    case PREMIUM = 'premium';
    case ADMIN = 'admin';
}

class User
{
    private UserType $type;

    public function __construct(UserType $type)
    {
        $this->type = $type;
    }

    public function isNormal(): bool
    {
        return $this->type === UserType::NORMAL; //
    }
}
```

Class Khusus untuk Konstanta

Buat class terpisah untuk mengelompokkan konstanta terkait:

```
final class UserTypes
{
    public const NORMAL = 'normal';
    public const PREMIUM = 'premium';
    public const ADMIN = 'admin';
}

class User
{
```

```
    private string $type;

    public function __construct(string $type)
    {
        $this->type = $type;
    }

    public function isNormal(): bool
    {
        return $this->type === UserTypes::NORMAL; //
    }
}
```

Dengan menggunakan pendekatan-pendekatan yang telah dijelaskan di atas untuk menghindari magic string, kita dapat memperoleh beberapa keuntungan signifikan dalam pengembangan perangkat lunak:

- IDE dapat memberikan autocompletion yang akurat dan cepat, membantu developer menulis kode dengan lebih efisien dan mengurangi kesalahan pengetikan
- Compiler memiliki kemampuan untuk mendeteksi kesalahan pengetikan secara otomatis pada tahap kompilasi, sehingga mencegah bug sebelum kode dijalankan
- Proses refactoring menjadi jauh lebih mudah dan aman karena semua nilai string terpusat di satu lokasi, memungkinkan perubahan dapat dilakukan secara konsisten di seluruh aplikasi
- Kode menjadi lebih self-documenting dan mudah dipahami oleh developer lain, karena penggunaan konstanta dan enum memberikan konteks yang jelas tentang maksud dan tujuan dari setiap nilai

Kirimkan Tugas Berat Ke Background

Pengenalan Laravel Queue

Laravel Queue memungkinkan kita untuk mengirimkan eksekusi tugas yang berat ke background process, sehingga aplikasi tetap berjalan lancar sambil memproses pekerjaan secara asinkron.

> **Note**
>
> Background process adalah proses yang berjalan di belakang layar tanpa mengganggu eksekusi utama aplikasi.

Dalam pengembangan aplikasi Laravel skala besar, kita sering membuat fitur yang membutuhkan waktu pemrosesan yang lama, misalnya pengiriman email, pemrosesan file, atau integrasi dengan layanan pihak ketiga. Jika dijalankan secara langsung, operasi-operasi ini bisa membuat aplikasi kita terasa lambat bagi pengguna.

Jika kita memiliki fitur untuk melakukan pembayaran lalu mengirimkan email mengenai status pembayaran. Tanpa queue, proses ini akan membuat server harus menunggu proses pengiriman email hingga selesai. Dengan queue, kita bisa memindahkan proses kirim email ke background, dibawah ini adalah ilustrasi dari penggunaan Queue.

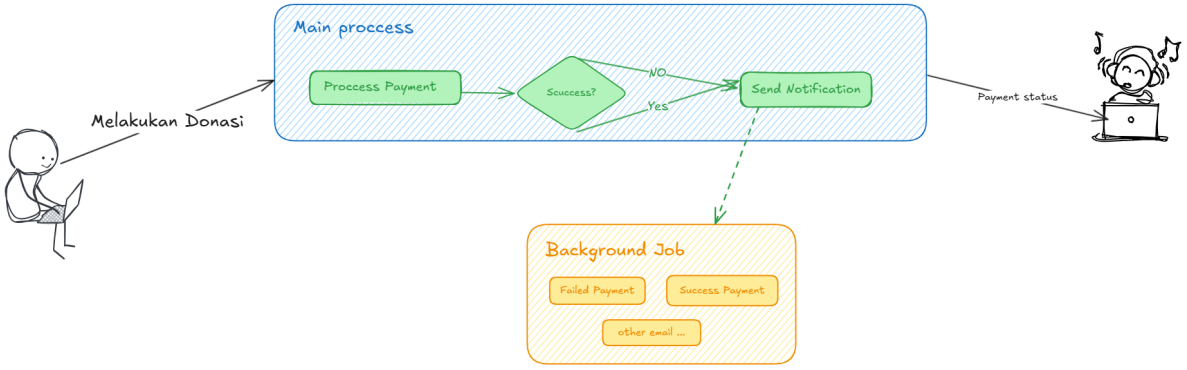

Figure 1: Laravel Queue

Jadi dari sisi user akan mendapatkan respon yang cepat karena server tidak perlu menunggu proses pengiriman email selesai. Queue akan menangani tugas pengiriman email di background sementara user bisa melanjutkan aktivitas lainnya di aplikasi.

Driver Queue

Laravel mendukung beberapa driver untuk menjalankan queue diantaranya adalah database, Redis, Amazon SQS, dan Beanstalkd. Masing-masing driver memiliki kelebihan dan karakteristik yang berbeda.

Database adalah pilihan paling sederhana karena tidak memerlukan service tambahan, Redis menawarkan performa tinggi dengan in-memory processing, sementara Amazon SQS cocok untuk aplikasi yang berjalan di AWS.

Implementasi

Contoh kali ini ada pada fitur pembayaran donasi, dimana setelah user melakukan pembayaran, sistem akan mengirimkan email notifikasi dan mengupdate status pembayaran. Untuk mengoptimalkan proses ini, kita akan menggunakan queue untuk menangani pengiriman email di background. Dengan begitu, user bisa langsung mendapatkan respons sukses pembayaran tanpa harus menunggu email terkirim.

Buat Background Job

Untuk membuat job baru di Laravel, kita bisa menggunakan perintah artisan:

```
php artisan make:job SendPaymentNotification
```

Perintah ini akan membuat class job baru di folder app/Jobs. Job class ini akan berisi logika untuk mengirim email notifikasi pembayaran.

Setelah job dibuat, kita perlu mengimplementasikan logika pengiriman email di dalam method handle(). Method ini akan dijalankan secara otomatis ketika job diproses oleh queue worker. Berikut adalah contoh implementasi job untuk mengirim email notifikasi pembayaran:

```
namespace App\Jobs;

use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendPaymentNotification implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;
```

```
    protected $payment;

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

    public function handle()
    {
        // Logika pengiriman email
        Mail::to($this->payment->user->email)
            ->send(new PaymentNotification($this->payment));
    }
}
```

Pada kode diatas kita melakukan implementasi dari berbagai macam fungsi yang ada dalam laravel, diantaranya adalah

ShouldQueue

`ShouldQueue` adalah interface yang menandakan bahwa job ini harus dijalankan dalam queue.

InteractsWithQueue

`InteractsWithQueue` adalah trait yang menyediakan metode untuk berinteraksi dengan antrian, seperti:

- Menghapus job dari queue
- Error Handling job
- Menunda eksekusi ulang job

Queueable

`Queueable` adalah trait untuk membuat antrian yang fleksibel seperti:

- Menentukan suatu job akan di eksekusi pada queue yang mana
- Menentukan koneksi queue, dengan ini kita bisa menentukan sartu job dijalankan dengan driver yang berbeda dengan lainnya
- Menentukan waktu delay sebelum job dieksekusi

Setelah membuat job class, kita bisa menjalankan job tersebut dari controller atau bagian lain aplikasi dengan cara:

```
SendPaymentNotification::dispatch($payment);
```

Secara otomatis, proses ini akan berjalan di background sehingga tidak menghambat respons aplikasi ke user. Diagram di bawah ini mengilustrasikan bagaimana flow queue bekerja dalam memproses job pengiriman email notifikasi pembayaran.

Cara Kerja Queue

Untuk mempermudah memahami proses penggunaan Queue anda dapat melihat diagram dibawah ini.

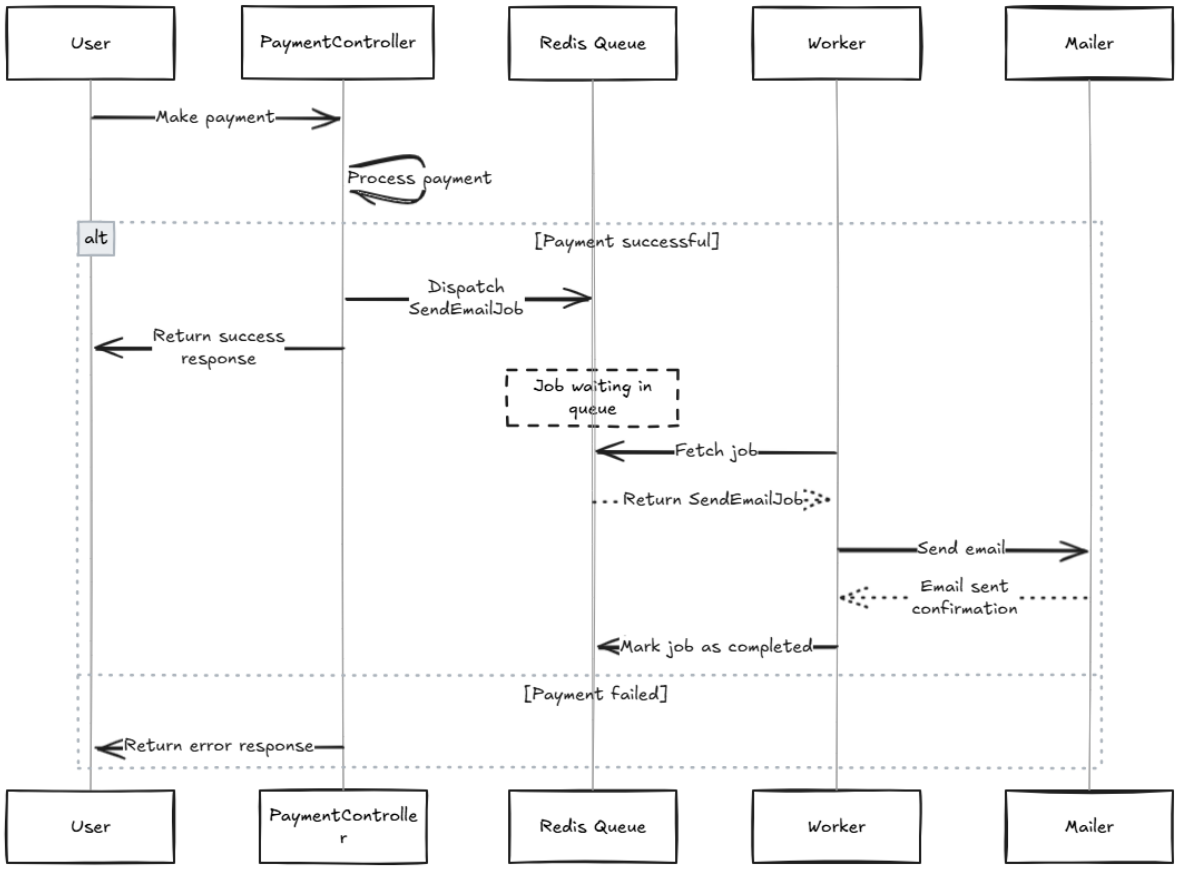

Figure 2: Cara Kerja Queue

Dari diagram tersebut, kita dapat melihat bahwa ketika sebuah job dikirim ke queue, queue worker akan memproses job tersebut secara asinkron. Job akan diambil dari queue satu per

satu, dieksekusi, dan hasilnya akan dicatat. Jika terjadi kegagalan, job dapat diulang sesuai dengan konfigurasi yang telah ditentukan.

Menjalankan Queue Worker

Untuk memproses job dalam queue, kita perlu menjalankan queue worker. Queue worker adalah proses yang berjalan di background dan terus memonitor queue untuk mengeksekusi job yang masuk. Untuk menjalankan queue worker, gunakan perintah:

```
php artisan queue:work
```

Di production, disarankan menggunakan process manager seperti Supervisor untuk memastikan queue worker tetap berjalan. Berikut contoh konfigurasi Supervisor:

> **Note**
>
> **Supervisor adalah** proses manajer yang digunakan untuk menjaga queue worker tetap berjalan secara otomatis.

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /path/to/artisan queue:work --sleep=3 --tries=3
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/path/to/worker.log
```

Dengan menggunakan queue aplikasi yang kita buat bisa lebih responsif memberikan response yang diberikan kepada user. Proses-proses berat seperti pengiriman email atau pemrosesan file dapat berjalan di background tanpa mengganggu pengalaman pengguna. Hal ini sangat penting terutama untuk aplikasi yang memiliki traffic tinggi.

Laravel Repository Pattern

*Design pattern*adalah suatu metode yang digunakan untuk menyelesaikan permasalahan yang sering terjadi dan biasanya memiliki suatu pola dalam menyelesaikan masalah.*Design Pattern*dapat mempercepat pengembangan suatu perangkat lunak. Salah satu dari*Design Pattern*yang paling sering digunakan adalah***Repository Pattern***.

What is Repository Pattern?

Secara singkatnya ***Repository Pattern*** adalah suatu pendekatan arsitektur perangkat lunak yang memisahkan antara *Business Logic layer* dengan *Data Access Logic layer*. Pemisahan ini bertujuan untuk membuat kode lebih terstruktur, mudah dikelola, dan mudah diuji.

Dengan menggunakan *repository pattern*, *business logic layer* tidak perlu mengetahui detail implementasi tentang sumber dan tujuan data. Ini berarti lapisan bisnis tidak perlu tahu apakah data berasal dari database SQL, NoSQL, API eksternal, atau sistem penyimpanan lainnya. *Business logic layer* hanya bertugas mengimplementasikan proses bisnis dan menyelesaikan masalah yang ada sesuai dengan kebutuhan aplikasi.

Pendekatan ini memberikan beberapa keuntungan penting: meningkatkan maintainability kode, memudahkan unit testing, dan membuat kode lebih fleksibel terhadap perubahan sumber data di masa depan. Selain itu, dengan memisahkan kedua lapisan tersebut, developer dapat bekerja secara paralel pada masing-masing lapisan tanpa saling mengganggu.

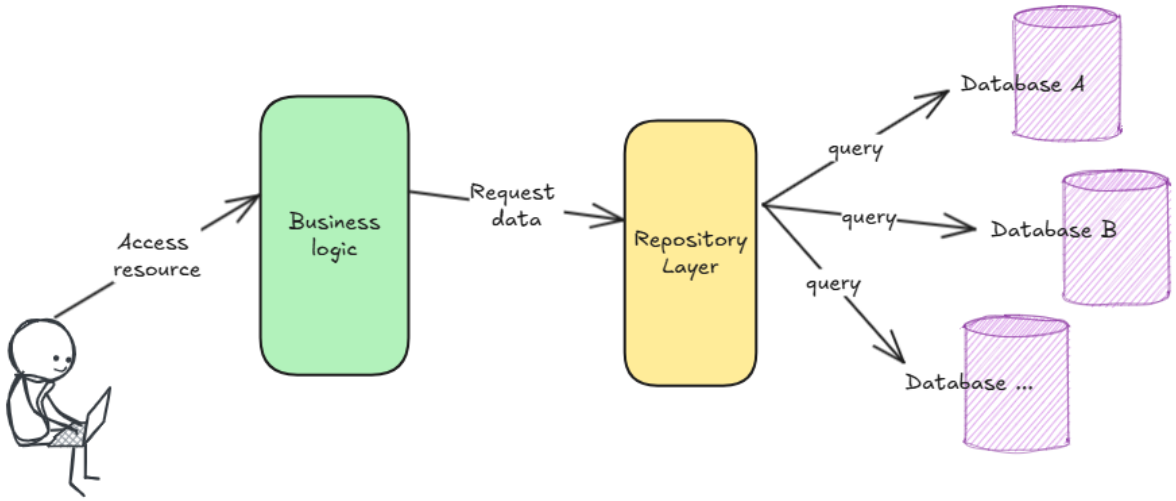

Figure 1: ilustrasi penggunaan repository pattern

Dalam implementasinya, *Repository Pattern* biasanya terdiri dari interface yang mendefinisikan operasi-operasi dasar untuk mengakses data (seperti Create, Read, Update, Delete) dan satu atau lebih implementasi konkret dari interface tersebut. Hal ini memungkinkan untuk mengganti implementasi repository tanpa mempengaruhi kode yang menggunakannya. Dengan pendekatan ini, kita dapat dengan mudah mengganti sumber data atau menambahkan fungsionalitas baru tanpa perlu mengubah logika bisnis yang ada.

Implementasi Repository Pattern

Pada contoh implementasi kali ini kita akan menggunakan tabel `campaigns` yang isinya adalah informasi iklan donasi yang ada pada sistem donasi online.

Buat Folder Khusus

Langkah pertama adalah kita membuat folder **Repository** folder ini akan menyimpan berbagai repository yang akan digunakan, folder ini berada didalam folder app, untuk lebih jelas perhatikan tree folder berikut ini

```
app
  Console
  Exceptions
  Http
      Controllers
      Middleware
```

```
Models
Providers
Repository --> Folder untuk menyimpan Repository
    Campaign
Utils
```

Buat Interface

Untuk memisahkan kode menjadi komponen yang independen (***code-decoupling***), kita perlu membuat sebuah ***interface***. Interface ini akan diimplementasikan oleh kelas-kelas yang membutuhkan *repository pattern*. *Interface* menjadi kontrak bagi *child class-nya* yang memberikan keleluasaan dalam pemilihan mekanisme *Data Access Logic*.

```php
<?php

namespace App\Repository\Campaign;

use Illuminate\Database\Eloquent\Collection;

interface CampaignRepositoryInterface
{
    public function getAll();
    public function findById(int $id);
    public function create(array $data);
    public function update(int $id, array $data);
    public function delete(int $id): bool;
}
```

Jika menggunakan Eloquent, kita dapat membuat kelas `EloquentCampaignRepositoy` sebagai *child* dari `CampaignRepositoryInterface`. Atau jika menggunakan Elasticsearch, kita dapat membuat *class* " ***`ElasticsearchCampaignRepository`*** *sebagai* child* dari `CampaignRepositoryInterface`.

Implementasi Interface

Setelah membuat interface, langkah selanjutnya adalah mengimplementasikan interface tersebut ke dalam sebuah kelas konkret. Dalam contoh ini, kita akan membuat implementasi menggunakan Eloquent ORM Laravel. Perhatikan kode berikut yang menunjukkan implementasi dari interface `CampaignRepositoryInterface`

```php
<?php

namespace App\Repository\Campaign;

class CampaignElaqunetRepository implements CampaignRepositoryInterface
{

    public function getAll()
    {
        return Campaign::all();
    }

    public function findById(int $id)
    {
        return Campaign::findOrFail($id);
    }

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

    public function update(int $id, array $data)
    {
        $campaign = Campaign::findOrFail($id);
        $campaign->update($data);
        return $campaign;
    }

    public function delete(int $id): bool
    {
        $campaign = Campaign::findOrFail($id);
        return $campaign->delete();
    }
}
```

Implementasi Controller

Setelah membuat repository dan interface-nya, langkah selanjutnya adalah mengimplementasikan repository tersebut di dalam controller. Berikut adalah contoh implementasi repository pattern di dalam controller:

```php
<?php

namespace App\Http\Controllers;

use App\Repository\Campaign\CampaignRepositoryInterface;

class CampaignController extends Controller
{
    private $campaignRepository;

    public function __construct(CampaignRepositoryInterface $campaignRepository)
    {
        $this->campaignRepository = $campaignRepository;
    }

    public function index()
    {
        $campaigns = $this->campaignRepository->getAll();
        return view('campaigns.index', compact('campaigns'));
    }

    public function show($id)
    {
        $campaign = $this->campaignRepository->findById($id);
        return view('campaigns.show', compact('campaign'));
    }
}
```

Dengan menggunakan dependency injection, controller tidak perlu mengetahui implementasi detail dari repository yang digunakan. Controller hanya perlu mengenal interface-nya saja, sehingga memungkinkan penggantian implementasi repository tanpa perlu mengubah kode pada controller.

Maka, dapat disimpulkan bahwa dengan menggunakan Repository Pattern, kita dapat memisahkan antara business logic dan data access logic dengan lebih baik. Repository Pattern mengabstraksi data access logic sehingga lebih mudah dikelola. Selain itu, pemisahan antara business logic dan data access logic juga mempermudah pengujian unit (unit testing) karena memungkinkan pembuatan mock objects untuk repository. Implementasi Repository Pattern membantu pengembangan perangkat lunak agar lebih sesuai dengan prinsip-prinsip SOLID. Dengan demikian, penggunaan Repository Pattern tidak hanya menyederhanakan tetapi juga memperlancar proses pengembangan perangkat lunak.

Logging dan Monitoring

Bagi developer pemula logging dan monitoring seringkali di lupakan, padahal hal ini merupakan suatu aspek yang penting. dengan ini kita bisa melakukan berbagai tindakan operasional yang bisa membuat aplikasi berjalan sesuai dengan fungsinya.

Aplikasi yang telah kita deploy ke production menunjukan penurunan performa, kasusnya adalah ketika mengambil data perlu loading time yang tinggi.

Maka ketika ada masalah tersebut kita bisa melakukan analisis log dan memerika apa yang sedang terjadi karena akan ada banyak kemungkinan yang menyebabkan aplikasi kita mengalami penurunan performa, dari hasil logging dan monitoring yang telah dilakukan kita bisa menentukan aksi selanjutnya yang lebih cepat dan terarah.

Apa yang dimaksud Logging dan Montoring?

Logging

Proses merekam aktivitas yang terjadi dalam suatu aplikasi dan menyimpannya ke dalam bentuk file atau sistem penyimpanan lainnya, biasanya disimpan dalam bentuk JSON atau dalam bentuk file .log lengkap dengan timestamp kejadian. Logging dapat digunakan untuk melakukan investigasi jika terjadi anomali pada aplikasi yang sedang berjalan.

Pada Laravel secara default semua log akan ditulis pada `/project/storage/logs/laravel.log` aktifitas yang di tulis adalah adalah ketika terjadi internal server error dan memberikan stack trace error tersebut. contohnya seperrti dibawa ini:

```
[2025-01-29 05:11:09] local.ERROR: SQLSTATE[08006] [7] connection to server at "127.0.0.1", p
password authentication failed for user "default" (Connection: pgsql, SQL: select c.relname a
space and n.nspname not in ('pg_catalog', 'information_schema') order by c.relname) at
/home/hellodit/project/php/beramal/vendor/laravel/framework/src/Illuminate/Database/Connectio
[stacktrace]

#0 project/php/beramal/vendor/laravel/framework/src/Illuminate/Database/Connection.php(767):
Illuminate\\Database\\Connection-> .... amework/src/Illuminate/Database/Connection.php(385):
Illuminate\\Database\\Connection->select()
#\
```

Informasi tersebut dapat kita gunakan untuk melakukan tindakan lanjutan seperti melakukan bugfixing.

Monitoring

Proses mengamati kondisi aplikasi yang berjalan secara realtime atau berkala, untuk memastikan aplikasi berjalan normal dan mendetaksi secara dini jika terjadi anomali. Proses monitoring biasanya menampilkan data dalam bentuk metrik, grafik atau dashboard contohnya seperti dibawah ini:

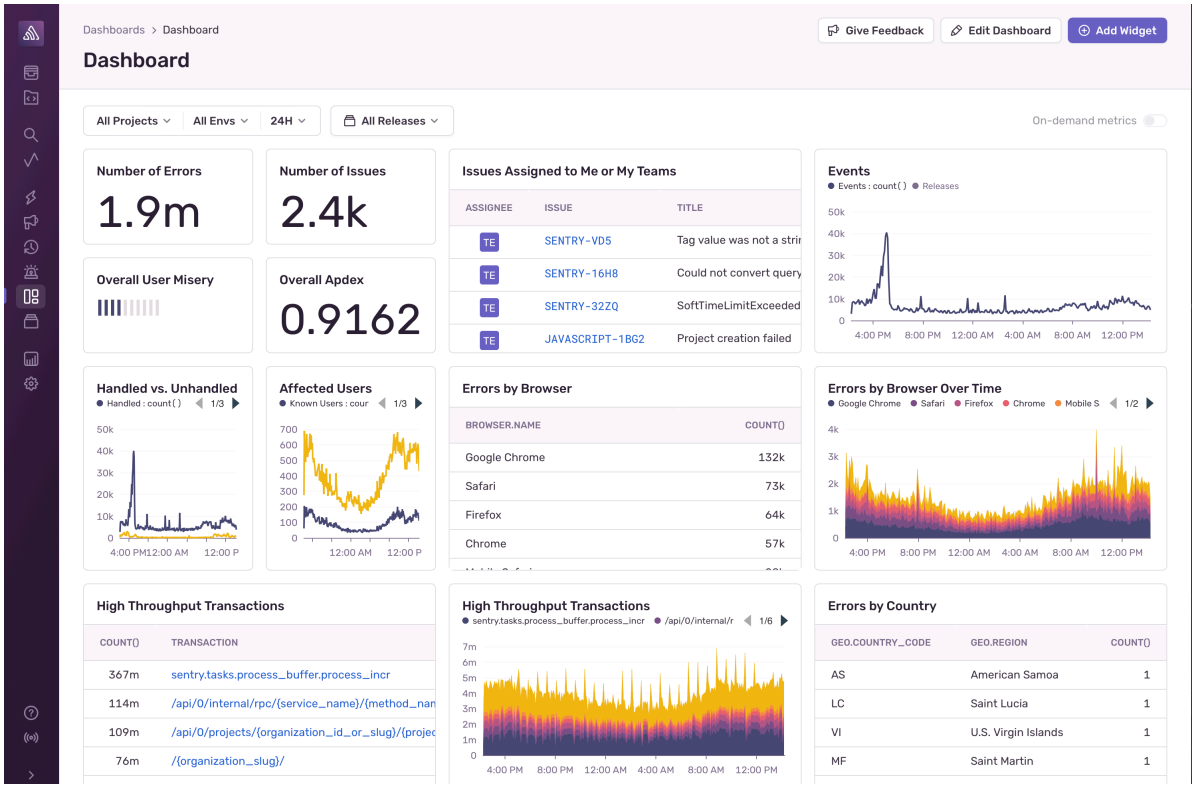

Figure 1: Dashboard Monitoring

Logging Aplikasi Laravel

Laravel menyediakan berbagai fitur logging yang dapat digunakan untuk mencatat aktivitas aplikasi. Secara default, Laravel menggunakan library Monolog yang menyediakan berbagai channel logging seperti single, daily, slack, dan lainnya. Konfigurasi logging dapat diatur melalui file config/logging.php.

Kita dapat mengatur level logging sesuai kebutuhan seperti debug, info, warning, error, critical, dan alert. Setiap level memiliki tingkat kepentingan yang berbeda dan dapat dikonfigurasi untuk diarahkan ke channel yang berbeda. Sebagai contoh, kita bisa mengirim error logs ke Slack untuk notifikasi tim, sementara debug logs disimpan dalam file lokal.

Berikut adalah beberapa contoh penggunaan level logging di Laravel:

```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

Setiap level logging memiliki fungsinya masing-masing dan sebaiknya digunakan sesuai dengan konteks dan tingkat urgensinya.

Best practice logging

Berikut adalah beberapa **Best practice** dalam menerapkan logging di aplikasi Laravel:

1. Gunakan log level yang sesuai - pastikan menggunakan level yang tepat sesuai dengan tingkat kepentingan informasi
2. Struktur log yang konsisten - format log harus konsisten dan mudah dibaca, idealnya menggunakan format JSON
3. Rotasi log file - terapkan rotasi log untuk menghindari file log yang terlalu besar dan memakan storage, pada .env laravel set configurasi berikut ini `APP_LOG=daily`
4. Informasi kontekstual - sertakan informasi yang membantu debug seperti timestamp, user ID, request ID
5. Hindari data sensitif - jangan mencatat informasi sensitif seperti password atau token dalam log

Contoh nya sebagai berikut

```
{
  "timestamp": "2025-02-23T12:34:56Z",
  "level": "error",
  "message": "Database connection failed",
  "context": {
    "exception": {
      "class": "PDOException",
      "message": "SQLSTATE[HY000] [1045] Access denied for user 'root'@'localhost'",
```

```
      "file": "/var/www/html/vendor/laravel/framework/src/Illuminate/Database/Connection.php"
      "line": 150
    },
    "request": {
      "method": "POST",
      "url": "/login",
      "ip": "192.168.1.1",
      "user_id": 12345
    }
  }
}
```

> **Note**
>
> Untuk memudahkan melihat isi log yang dihasilkan oleh aplikasi, kita bisa menggunakan package opcodesio/log-viewer

Selain itu, penting untuk melakukan monitoring terhadap ukuran log file dan mengimplementasikan strategi pembersihan log yang efektif. Log yang terlalu besar dapat mempengaruhi performa sistem dan menyulitkan proses analisis.

Rotasi FIle Log

Untuk mengatasi hal ini, sebaiknya menerapkan strategi rotasi log yang teratur dan menyimpan log historis di storage terpisah. Implementasi tools monitoring seperti ELK Stack atau Graylog juga dapat membantu dalam mengelola dan menganalisis log secara lebih efektif.

Selain itu, penggunaan log rotation dan monitoring tools juga membantu dalam mengoptimalkan penggunaan storage dan mempermudah proses analisis log. Dengan menerapkan praktik logging yang baik, developer dapat lebih mudah melakukan troubleshooting dan maintenance aplikasi.

Untuk memastikan efektivitas logging, penting juga untuk melakukan review berkala terhadap log yang dihasilkan dan memastikan bahwa informasi yang dicatat benar-benar berguna untuk proses debugging dan analisis. Implementasi logging yang baik harus seimbang antara detail informasi yang dicatat dan overhead yang dihasilkan pada sistem.

Monitoring Aplikasi Laravel

Untuk melakukan monitoring aplikasi laravel yang telah kita buat, tersedia banyak tools yang bisa digunakan, berikut adalah beberapa tools yang sering digunakan untuk melakukan mon-

itoring pada aplikasi Laravel. Tools-tools ini memiliki keunggulan masing-masing dan dapat disesuaikan dengan kebutuhan project yang sedang dikerjakan.

Laravel telescope

Laravel Telescope adalah package debugging yang powerful untuk aplikasi Laravel. Package ini menyediakan insight mendalam tentang request yang masuk, exception, query database, queued jobs, mail, notifikasi, cache operations, scheduled tasks, variable dumps dan banyak lagi. Telescope membantu developer memahami apa yang terjadi di dalam aplikasi secara real-time dan mempermudah proses debugging. Informasi lebih lengkap dapat klik link berikut ini.

Health Checkpoint

Health checkpoint adalah fitur yang memungkinkan kita untuk memonitor kesehatan aplikasi dengan melakukan pengecekan terhadap komponen-komponen kritis seperti koneksi database, cache, queue, dan layanan eksternal. Dengan mengimplementasikan health checks, kita dapat mendeteksi masalah sebelum berdampak pada pengguna.

Kita bisa membuat sendiri atau menggunakan package seperti `spatie/laravel-health` yang dapat digunakan untuk mengimplementasikan health checks dengan mudah.

Application Performance Monitoring

Beberapa tools APM (Application Performance Monitoring) populer yang dapat digunakan untuk memonitor aplikasi Laravel, contohnya Sentry, New Relic, Datadog, dan Scout APM.

> **Note**
>
> Gunakan Sentry unutk tools APM yang ramah pemula, dan terdapat free tier untuk belajar.

Tools ini menyediakan insight mendalam tentang performa aplikasi, memungkinkan developer untuk mengidentifikasi bottleneck, melacak response time, dan menganalisis penggunaan resources.

Dengan menggunakan APM tools, tim dapat proaktif mendeteksi dan menyelesaikan masalah performa sebelum mempengaruhi pengalaman pengguna.

Kesalahan Umum Logging dan Monitoring

Terlalu banyak log

Logging terlalu banyak informasi yang tidak diperlukan dapat menyebabkan overhead pada sistem dan mempersulit proses analisis. Sebaiknya fokus pada informasi yang benar-benar

penting dan relevan untuk debugging dan monitoring. Pastikan untuk menyaring data yang dicatat agar log tetap ringkas dan bermakna.

Tidak peduli log level

Menggunakan log level yang tidak tepat atau mengabaikan penggunaan level yang berbeda dapat menyebabkan kesulitan dalam mengidentifikasi dan memprioritaskan masalah. Penting untuk memahami kapan menggunakan level debug, info, warning, atau error sesuai dengan konteksnya. Penggunaan log level yang tepat membantu dalam proses troubleshooting dan memudahkan dalam memfilter log berdasarkan tingkat kepentingannya.

Tidak manage file log

Jika memiliki storage yang terbatas, file log bisa menjadi salah satu yang memakan banyak storage, maka dari itu perlu dilakukan hapus seara berkala dalam kurun waktu tertentu. Hal ini bisa menghemat storage dan memudahkan tools melakukan analisis file log.

N+1 Query Problem

Secara sederhana n+1 adalah problem yang seringkali terjadi ketika aplikasi melakukan query ke database, problemnya terletak pada jumlah query yang berlebihan, sehingga menurunkan performa dari aplikasi yang sedang berjalan.

N+1 Sering terjadi ketika aplikasi yang kita buat menggunakan ORM, pada konteks Laravel N+1 sering terjadi ketika kita menuliskan query dengan menggunakan Elaqunent ORM

Studi kasus N+1 Query Problem

Contoh Masalah

Pada aplikasi yang kita buat memiliki tabel `Campaigns` dan `Comments`, setiap campaign bisa memiliki banyak comments, pada suatu halaman kita akan menampikan data campaigns dan comments yang terkait. Maka seringkali melakukan query dengan menggunakan syntax seperti dibawah ini

```
// CampaignController -> index
$campaigns = Campaigns::where('status','=','active')->get();

foreach ($campaigns as $campaign) {
    echo $campaign->title . ":\n";
    // Setiap kali mengakses $campaign->comments, query baru dijalankan:
    // SELECT * FROM comments WHERE campaign_id = ?
    foreach ($campaign->comments as $comment) { // N query untuk mengambil comments
        echo "- " . $comment->content . "\n";
    }
}
```

Pada query pertama adalah `SELECT * FROM campaigns WHERE status = 'active'` lalu untuk setiap campaigns aplikasi melakukan query kembali untuk menampilkan data comments yang terkait dengan campaign, query yang dilakukan adalah `SELECT * FROM comments WHERE campaign_id = ?`

Misalknya terdapat data 100 Campaign maka query yang dilakukan adalah 101 Query, dengan penjelasan

- Query utama: 1 untuk menggambil data Campaign dan menyimpannya pada variable `$campaigns`
- Query tambahan: 100 masing - masing untuk mengambil data Commments yang terkait

Untuk memudahkan pemahaman mengenai N+1 Query kamu bisa melihat diaragm dibawah ini:

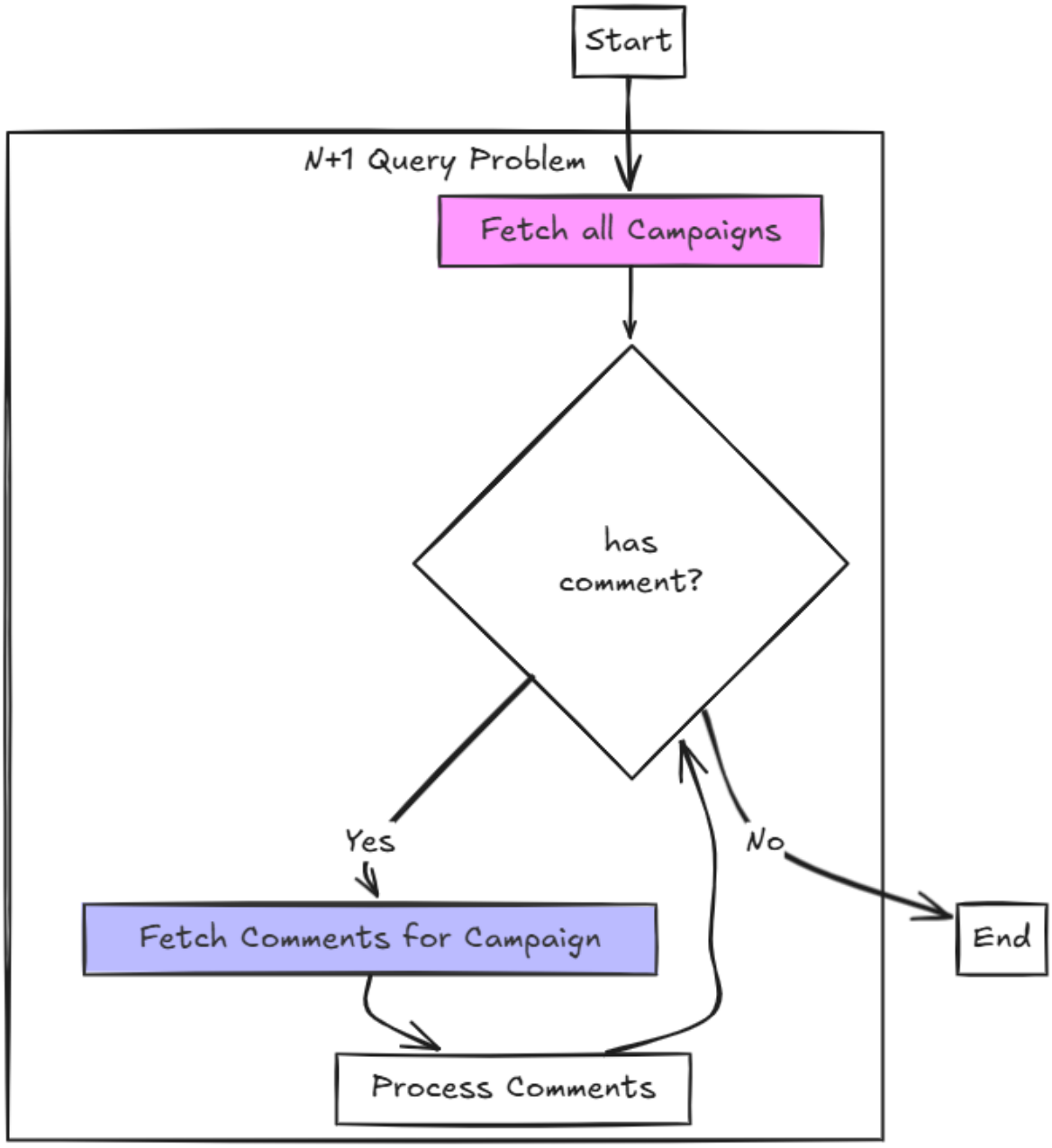

Figure 1: N+1 Query Problem

Solusi utama: Eager Loading

Solusi N+1 Query pada Laravel adalah **Eager Loading,** dengan menggunakan eager loading maka seluruh data akan diambil dalam satu query utama. Eager loading memungkinkan kita untuk mengambil data pada tabel yang saling berhubungan, misalnya tabel Campaigns dan Comments.

Untuk menerapkan eager loading maka hal yang perlu dilakukan adalah mendefinisikan relation pada masing - masing model tabel yang saling berhubungan, misalnya:

```
// campaigns model
class Campaign extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}

// comments model
class Comment extends Model
{
    public function campaign()
    {
        return $this->belongsTo(Campaign::class);
    }
}
```

Kata kunci utama untuk mengaktifkan eager loading pada tiap query yang dibuat adalah `with()` , berikut ini adalah contoh implementasi.

```
// CampaignController -> index
$campaigns = Campaigns::where('status','=','active')->with('comments')->get();

foreach ($campaigns as $campaign) {
    echo $campaign->title . ":\n";
    // Data comments sudah ada di memori, tidak ada query baru
    foreach ($campaign->comments as $comment) {
        echo "- " . $comment->content . "\n";
    }
}
```

Dengan kode diatas maka query yang dihasilkan adalah:

1. `SELECT * FROM campaigns where status = ‘’active";`

   Query ini mengambil data utama campign, misal terdapat 100 active campaigns, maka akan mendapatkan ID: 1,2,3,4,5 … 100

2. `SELECT * FROM comments WHERE campaign_id IN (1, 2, 3, ..., 100);`

   Query ini akan menggambil semua data komentar untuk 100 campaigns sekaligus, hasilnya dalam memory sudah terdapat data 100 campaigns dan semua komentar yang terkait dengan campaigns

Perbandingan Jumlah Query

JIka dibandingkan maka berikut ini adalah jumlah query dari masing - masing metode yang telah kita implementasikan

Metode	Query Utama	Query Tambahan	Total Query
Tanpa Eager Loading	1	100	101
Dengan Eager Loading	1	1	2

Maka dapat kita simpulkan bahwa:

- **Tanpa Eager Loading**, aplikasi melakukan**11 query**untuk mengambil 10`Campaign`dan`Comment`terkait.
- **Dengan Eager Loading**, aplikasi hanya melakukan**2 query**untuk mengambil data yang sama.
- Eager Loading secara signifikan mengurangi jumlah query, meningkatkan performa aplikasi, dan mengurangi beban pada database.

Bagaimana Data Comments Dipetakan ke Campaign?

Setelah ke 2 query dijalankan, Laravel Elaquent akan secara otomatis memetakan ke data comments ke Campaigns yang sesuai menggunakan campaign_id pada tabel comments. Proses ini disebut dengan **data Hydration**

Fitur Lanjutan Eager Loading di Laravel

Eager Loading Multiple Relationships

Kita bisa melakukan eager loading pada beberapa relasi sekaligus, ini sangat berguna untuk tabel yang memiliki relasi lebih dari 1 ke tabel lainnya, misalkan tabel `campaigns` dengan tabel `comments` dan tabel `user` contoh penulisannya adalah sebagai berikut:

```
$campaigns = Campaign::with(['comments', 'user'])->get();
```

Nested Eager Loading

Seringkali kita ingin mengambil data dari relasi yang berelasi, misal kita ingin menggambil data author dari tabel comments, maka kita bisa menulis sintax seperti dibawah ini:

```
$campaigns = Campaign::with('comments.user')->get();
```

Constraining Eager Loads

Seringkali kita ingin melakukan eager loading pada suatu relasi dan menambahkan kondisi pada query yang kita buat, hal ini bisa kita lakukan dengan memberikan array ke method `with()` .

Misalnya kita hanya ingin menampilkan Campaign yang memiliki komentar lebih dari 10, maka kita bisa menggunakan query seperti dibawah ini

```
$campaigns = Campaigns::where('status','=','active')
            ->with(['comments' => function (Builder $query) {
                    $query->count() > 10;
                }])->get();
```

Kesimpulan

Eager Loading sangat berguna untuk melakukan akses data yang berelasi karena memungkinkan pengambilan data secara efisien dalam jumlah query yang minimal. Dengan teknik ini, Anda tidak hanya menghemat sumber daya database tetapi juga meningkatkan responsivitas aplikasi secara signifikan Eager Loading wajib dipakai untuk membuat aplikasi lebih scalable