SPASI

30 TIP KODING LARAVEL YANG LEBIH MANUSIAWI DAN DISUKAI REKAN KERJAMU

BAYU HENDRA WINATA

Daftar Isi

View

View merupakan huruf kedua dari akronim M-V-C. Berbeda dengan Model dan Controller yang berisi kode PHP, di View kita akan lebih banyak berurusan dengan HTML dan teman-temannya. Istilah kerennya, View adalah *presentation layer*, yaitu suatu bagian yang tugasnya melakukan presentasi (menyampaikan informasi) ke pengguna aplikasi.

Sebuah View yang *clean* sama pentingnya dengan Controller dan Model yang *clean*. Bahkan View harusnya lebih mendapat prioritas karena "merapikan" View jauh lebih mudah dilakukan dibanding merapikan Controller atau Model.

Yuk, kita buktikan!

Memecah File

Salah satu kemampuan yang harus dikuasai untuk menulis kode yang *friendly* adalah keberanian untuk memecah kode atau *file*.

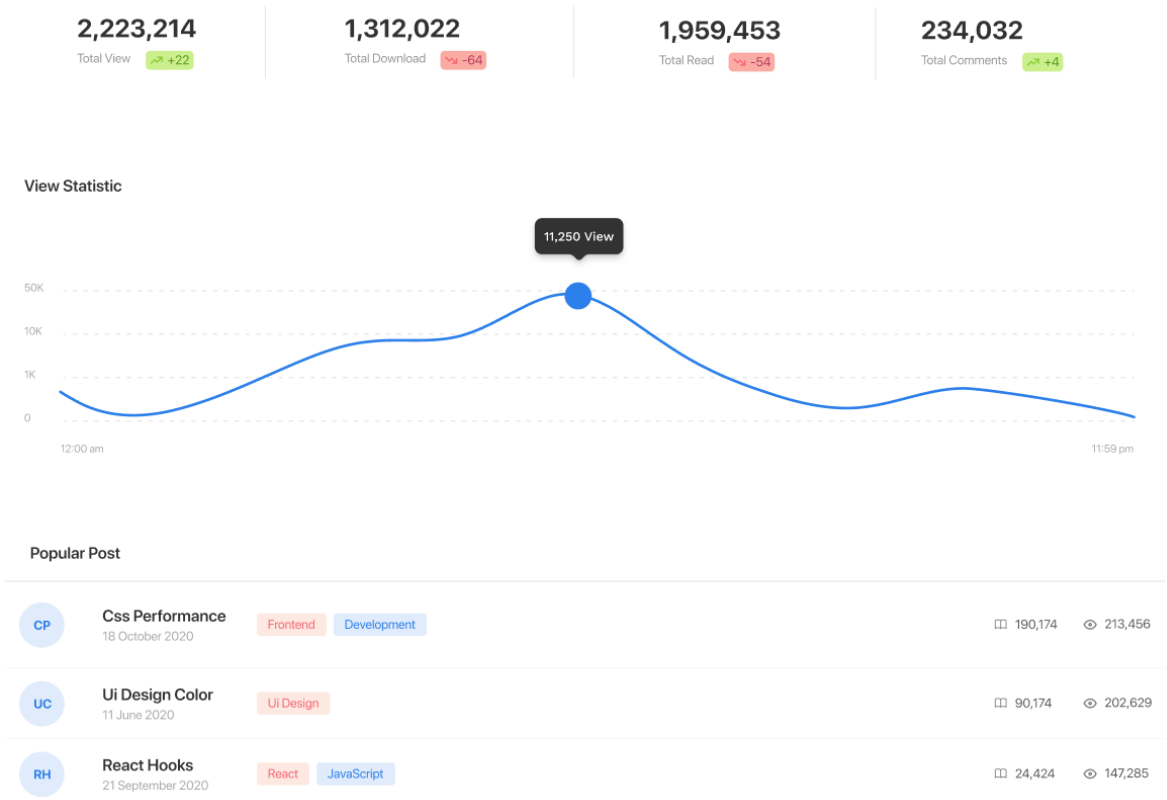
Di awal proyek, semua masih terlihat rapi. Kodenya masih sedikit. Seiring berjalannya waktu, ada penambahan fitur disana-sini, tambal sulam *bug* di kanan dan di kiri. Kode yang awalnya masih terlihat dalam satu layar sekarang harus di-*scroll* berkali-kali untuk melihat keseluruhan isinya.

Programmer yang baik tahu kapan harus berhenti sejenak, mendeteksi bagian mana yang mulai membengkak dan berpotensi menyulitkan untuk dibaca di kemudian hari, lalu mulai memecahnya. View sengaja dijadikan topik pertama karena memecah view paling mudah dilakukan dan hampir tidak ada efek sampingnya.

> Keuntungan memecah baris kode yang besar menjadi beberapa file kecil akan semakin terasa berlipat ganda ketika kamu bekerja dalam sebuah tim yang menerapkan *version control system* seperti Git.

Biasakan Memakai Sub View

Bayangkan kamu mendapat tugas untuk membuat dashboard dengan mockup seperti di bawah ini.



Pada umumnya, tampilan di atas akan diimplementasi menjadi satu file blade seperti berikut:

```
@extends('layout')

@section('content')
    <h1>Statistik Laporan</h1>
    <section>
        ...
        Summary
        ...
    </section>
    <section>
        ...
        Chart
        ...
    </section>
    <section>
        ...
        List of popular post
        ...
    </section>
@endsection
```

Sekarang mari kita coba untuk memecahnya menjadi *sub view*. Bagaimana caranya?

Secara kasat mata, kita bisa melihat ada tiga komponen utama yang menyusun halaman dashboard di atas, yaitu:

1. *Summary*
2. *Chart*
3. *List of popular post*

Setelah mengetahui komponen penyusun halaman dashboard tersebut, langkah berikutnya adalah membuat ***sub view*** untuk masing-masing komponen:

1. `_summary.blade.php`
2. `_chart.blade.php`
3. `_popularPost.blade.php`

Lalu, kamu cukup memanggil tiap komponen dengan **@include**:

```
@extends('layout')

@section('content')

    <h1>Dashboard</h1>
    @include('_summary')
    @include('_chart')
    @include('_popularPost')
@endsection
```

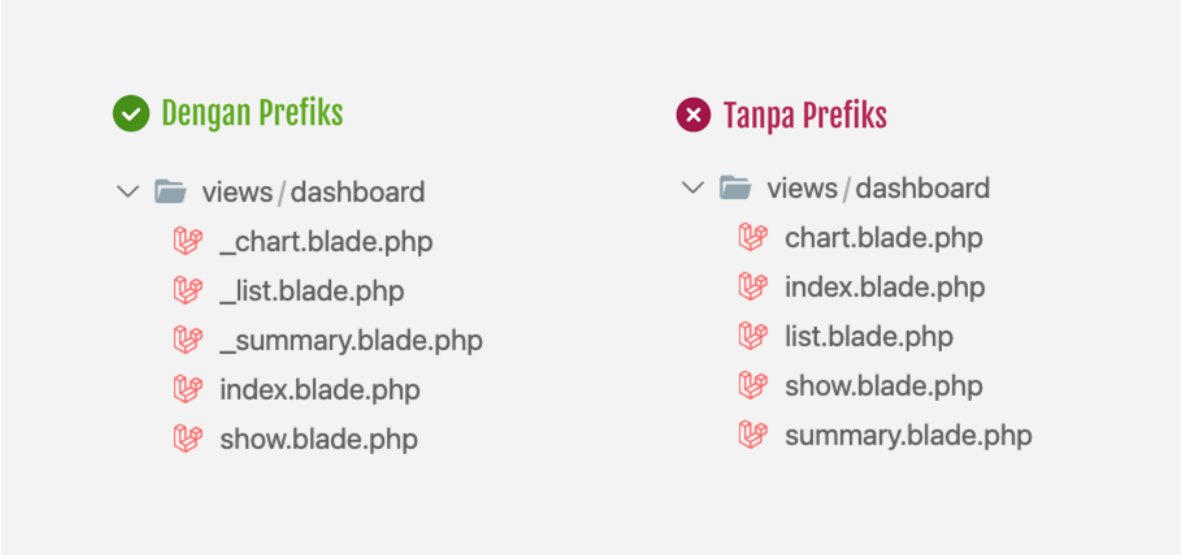
Alih-alih punya satu *file* yang berisi 100 baris kode, sekarang kamu punya 4 *file* yang masing-masing berisi 25 baris kode. Lebih rapi dan lebih ringan ketika dibuka di *code editor* atau IDE.

> Kode yang baik adalah kode yang mencerminkan kebutuhan fungsional aplikasinya. Maksudnya adalah ketika kita bilang ada fitur a, b, dan c di aplikasi, maka a, b, dan c itu idealnya juga terlihat secara eksplisit di kode penyusun aplikasi, entah itu sebagai nama *file*, nama fungsi, atau nama Class.

Penamaan Sub View

Kamu mungkin bertanya kenapa *file blade* pada contoh sebelumnya diberi nama `_summary.blade.php` (perhatikan ada *underscore* diawalnya) dan bukan `filter.blade.php` saja.

Dengan menambahkan *underscore* sebagai prefiks, maka kita bisa melihat dengan jelas mana *view* utama dan mana *sub view*. Editor yang kamu pakai secara otomatis akan mengurutkan *file* secara alfabetis dan seolah-olah mengelompokkan *file* menjadi dua bagian: bagian atas untuk *sub view* dan bagian bawah untuk *view* utama.



Secara sekilas kita bisa melihat bahwa *sub view* yang diberi prefiks lebih mudah dikenali dibanding yang tanpa prefiks. *Minimum effort, maximum effect*.

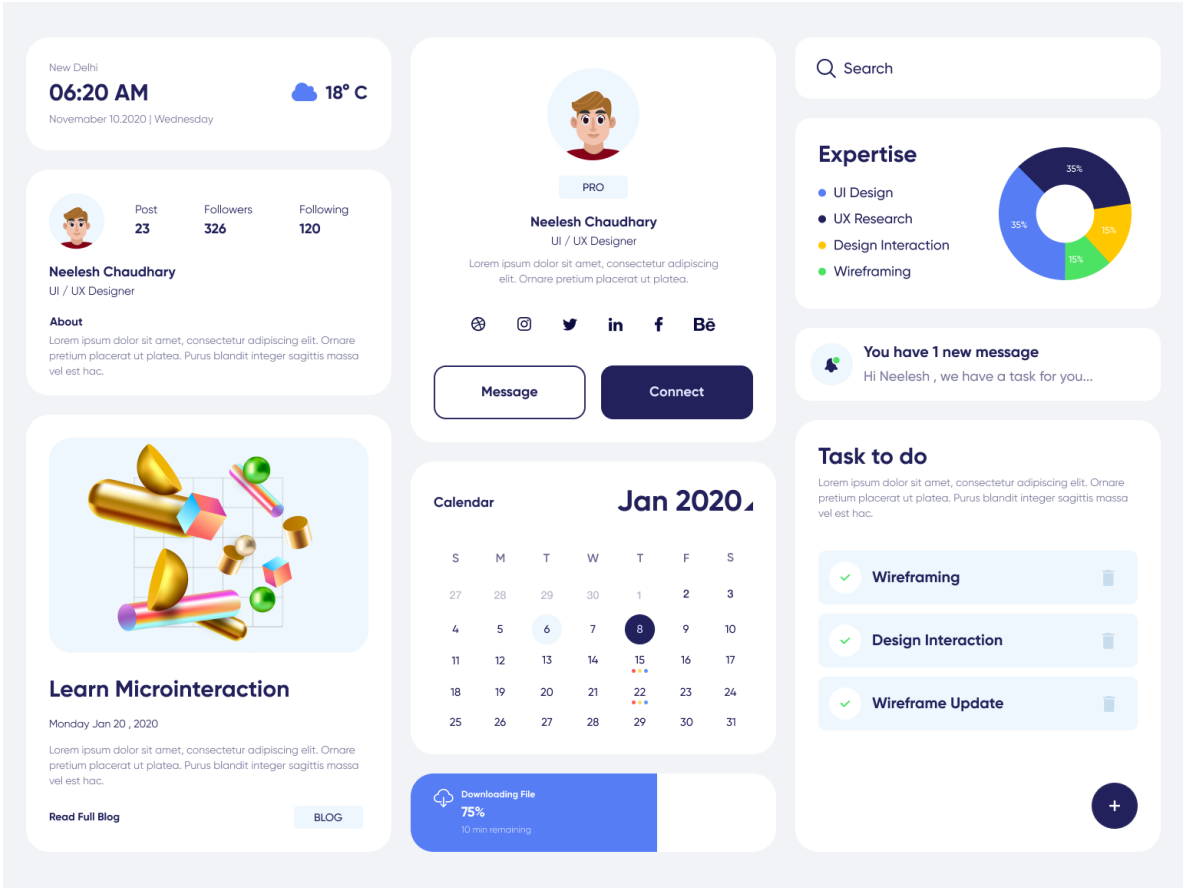
Lebih jauh lagi, kamu juga bisa membuat folder baru untuk meletakkan *sub view*. Nama yang umum dipakai biasanya **partials** atau **sub**. Kalau sudah dibuatkan folder khusus untuk menampung *sub view*, maka nama filenya tidak perlu lagi diberi prefiks "_" (*underscore*).

> Ingat prinsipnya, **kelompokkan yang sejenis**.

Layout vs Konten

Setelah paham kapan harus mulai memecah *view* agar tidak membengkak, selanjutnya kita perlu paham **dimana** sebuah *view* harus dipecah. Terkadang ~~gambar~~ kode bisa menggantikan 1000 kata, jadi mari kita lihat contohnya.

Anggap kita sedang mengerjakan aplikasi menggunakan Bootstrap. Lalu tampilan yang ingin dibuat adalah seperti di bawah ini:



Jika tidak hati-hati, maka *view* yang kamu buat akan seperti ini:

```
<div class="container">
  <div class="row">
    @include('_weather')
    @include('_profile-stat')
    @include('_blog')
    @include('_profile-full')
    @include('_calendar')
    @include('_searchbox')
    @include('_expertise')
    @include('_inbox')
    @include('_todo')
  </div>
</div>
```

Apa yang salah dari kode di atas?

Kita kehilangan informasi tentang susunan grid. Melihat kode seperti di atas, susah untuk membayangkan bagaimana hasil *rendering* halaman tanpa melihat langsung di *browser* atau melihat satu persatu isi setiap *sub view*.

Cara yang lebih baik adalah dengan meng-**eksplisit**-kan struktur grid di *view* utama.

```
<div class="container">
  <div class="row">
    <div class="col-sm-4">
        @include('_weather')
        @include('_profile-stat')
        @include('_blog')
    </div>
    <div class="col-sm-4">
        @include('_profile-full')
        @include('_calendar')
    </div>
    <div class="col-sm-4">
        @include('_searchbox')
        @include('_expertise')
        @include('_inbox')
        @include('_todo')
    </div>
  </div>
</div>
```

Sekarang terlihat dengan jelas bahwa halaman di atas terbagi menjadi 3 kolom. Dengan meletakkan tag HTML untuk *layouting* di *view* utama, kamu bisa mengganti susunan layout dengan sangat mudah. Cukup utak-atik posisi **@include**. Sub view tidak perlu diubah.

> ***View* utama untuk mengatur layout, *sub view* untuk merender konten**. Ketika melihat *view* utama, pastikan kamu bisa membayangkan bagaimana layout halamannya.

Tidak Menyisipkan Blade Ke Dalam Javascript

Mengembangkan aplikasi web tidak bisa lepas dari Javascript. Begitu juga dengan aplikasi Laravel yang dikembangkan secara *fullstack*.

```
<section>
    <button id="buttonSubmitComment">Kirim Komentar</button>
</section>

<script>
$('#buttonSubmitComment').on('click', function(e) {
    e.preventDefault();
    $.ajax({
        url: {{ route('comment.store') }},
        type: "POST",
        dataType: 'json',
    })
});
<script>
```

Mengoplos kode PHP dan Javascript seperti contoh di atas setidaknya memiliki dua kekurangan:

1. Membaca 2 *syntax* dari 2 bahasa yang berbeda dalam satu blok kode yang sama akan sedikit merepotkan otak (*context switching*) dan berpotensi menimbulkan kesalahan dasar ketika menulisnya.

   ```
   url: {{ route('comment.store') }},
   ```

   Berapa detik yang kamu butuhkan untuk menyadari bahwa potongan kode di atas salah secara sintaksis?

2. Jika suatu ketika kamu ingin memindahkan semua *script* dari file Blade ke satu file `js`, maka tidak bisa dilakukan secara langsung karena fungsi `route()` tidak akan

dikenali di file `js`. Harus di-*refactor* dulu.

Ketika kebutuhan aplikasi mengharuskan adanya interaksi antara kode Blade(PHP) dan Javascript, ada dua cara yang bisa dilakukan agar hubungan tersebut bisa langgeng dalam jangka panjang (mudah di-*maintain*):

1. Passing sebagai data-attribute
2. Definisikan *dynamic variable* di awal kode

> Sekedar mengingatkan, kode `url: {{ route('comment.store') }}` di atas salah karena kurang tanda petik. Kode yang benar seharusnya `url: "{{ route('comment.store') }}"`

Passing Variable Sebagai Atribut HTML data-*

Alih-alih mencampur Blade dan Javascript, kamu bisa memanfaatkan atribut HTML `data-*` untuk mem-*passing* sebuah value yang berasal dari PHP agar bisa dibaca oleh Javascript.

```
<section>
    <button id="buttonSubmitComment" data-url="{{ route('comment.store') }}">Kirim
Komentar</button>
</section>

<!-- End of Blade here -->
<!-- Dari baris ini ke bawah khusus Javascript -->

<script>
$('#buttonSubmitComment').on('click', function(e) {
    e.preventDefault();
    $.ajax({
        url: $(this).data('url'),
        type: "POST",
        dataType: 'json',
    })
});
<script>
```

Atribut `data-*` merupakan atribut HTML5 yang valid digunakan untuk semua elemen. Artinya kamu bisa menambahkan `data-*` ke `<form>`, `<button>`, `<table>`, dan semua tag HTML lain.

Cara mengaksesnya juga sangat mudah.

```
// Dengan Javascript native
document.querySelector('#buttonSubmitComment').dataset.url;

// Dengan jQuery
$('#buttonSubmitComment').data('url');
```

Dengan metode penulisan seperti di atas, kamu telah berhasil menjauhkan diri dari ***kode oplosan***, yaitu suatu kondisi bercampurnya 2 bahasa dalam **satu blok kode**.

Sekilas Tentang *Context Switching*

Context switching adalah sebuah kondisi ketika kita harus berpindah dari satu aktivitas ke aktivitas lain.

Dilihat dari kaca mata *resources*, *context switching* itu mahal. Berpindah dari mode PHP ke mode Javascript juga sama. Oleh sebab itu penting bagi kita untuk bisa mengelompokkan masing-masing kode ke dalam "blok"-nya masing-masing.

Contohnya sama seperti saat kamu membaca buku ini. Setiap selesai satu bagian kamu pegang *hape*, buka notififikasi, membalas komentar, lalu kembali melanjutkan membaca buku. Ada sekian detik waktu tambahan yang dibutuhkan otak kita untuk kembali fokus ke aktivitas membaca buku.

> ***Context switching*** dalam waktu yang singkat dengan intensitas yang tinggi sangat mengganggu produktivitas dan dan tidak baik untuk kesehatan mental. Hindarilah semaksimal mungkin!
>
> Referensi: https://blog.rescuetime.com/context-switching/

Definisikan *Dynamic Variable* Di Awal Kode

Jika karena suatu hal metode sebelumnya tidak bisa diterapkan, maka opsi lainnya adalah dengan mendefinisikan semua variabel di awal dengan *keyword* `let` ataupun `const`.

```
<section>
    <button id="buttonSubmitComment">Kirim Komentar</button>
</section>


<script>

<!-- Area transisi, serah terima antara Blade dan Javascript -->
const URL = '{{ route('comment.store') }}';
const sampleData = @json($dataFromController);

<!-- Setelah ini full Javascript, tidak ada lagi oplosan -->
$('#buttonSubmitComment').on('click', function(e) {
    e.preventDefault();
    $.ajax({
        url: URL,
        type: "POST",
        dataType: 'json',
    })
});
<script>
```

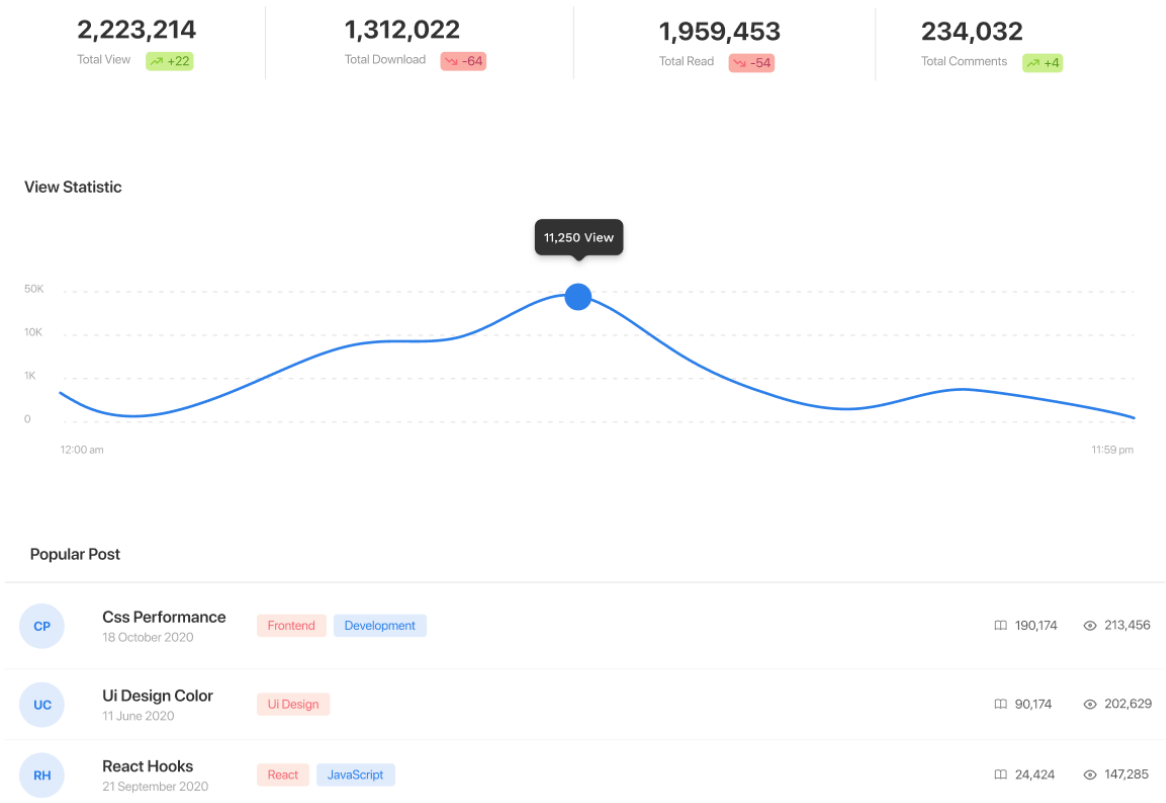
Sekali lagi, kata kuncinya adalah **pengelompokkan**. Sekarang kita punya satu blok kode yang khusus menjadi tempat perantara antara PHP dan Javascript. Kurang ideal, tetapi tetap lebih rapi dibanding membiarkan kode PHP bercampur dengan Javascript, berserakan di setiap baris.

Fullstack application merujuk ke aplikasi yang *backend* dan *frontend* tergabung dalam satu *codebase*. Alternatifnya, *backend* memiliki *codebase* sendiri (misalnya memakai Java) dan *frontend* memiliki *codebase* sendiri (misalnya memakai Vue.js).

Jangan Pisahkan JS dan Pasangan HTML-nya

Di bagian sebelumnya, kita sudah mengenal cara memecah satu file View yang besar menjadi beberapa *sub view* yang kecil. Nah, kamu harus berhati-hati ketika melakukan pemecahan tersebut. Pastikan JS dan HTML yang saling berhubungan tetap berada dalam satu file yang sama.

Mari kita lihat kembali contoh mockup dashboard sebelumnya:



Sekarang ada kebutuhan, setiap kali Popular Post diklik akan muncul modal yang berisi statistik sesuai judul yang dipilih.

Kode yang ditulis biasanya seperti di bawah ini:

```
@extends('layout')

@section('content')

    <h1>Dashboard</h1>
    @include('_summary')
    @include('_chart')
    @include('_popularPost')
@endsection

@push('script')
    <script>
        $('#listItem .item').on('click', function() {
          //show modal
        });
    </script>
@endpush
```

Tidak ada masalah buat yang menulis kode. Tapi akan menimbulkan pertanyaan bagi yang membacanya. Ada di mana tag HTML yang mempunyai id `#listItem` ?

Ingat, ketika programmer melakukan *debugging* asal sebuah variabel, urutan yang dilakukan biasanya:

1. Mencari di blok kode atau fungsi yang sama (apa yang ada di dalam blok `<script>`).
2. Mencari dalam file yang sama (`dashboard.blade.php`).
3. Mencari dalam folder yang sama (`resources/views`).
4. Menyerah, mari cari di semua folder dengan fitur "Find" bawaan editor.

Mungkin bukan masalah besar. Pada akhirnya akan ketemu juga. Tapi akan sedikit mengurangi *happiness index* ketika koding.

Lalu bagaimana solusi yang lebih manusiawi?

1. Dekatkan Yang Saling Membutuhkan

Untuk setiap sub view membutuhkan kode Javascript, maka kodenya cukup ditulis di masing-masing sub view tersebut.

```
<!-- dashboard.blade.php -->
@extends('layout')

@section('content')

    <h1>Dashboard</h1>
    @include('_summary')
    @include('_chart')
    @include('_popularPost')
@endsection

<!-- _popularPost.blade.php -->
<section id="popularPost">
    @foreach($popularPost as $post)
    <div class="item">...</div>
    @endforeach
</section>

@push('script')
    <script>
        $('#popularPost .item').on('click', function() {
          //show modal
        });
    </script>
@endpush

<!-- _summary.blade.php -->
<section id="summary"></section>

@push('script')
    <script>
        // Script lainnya disini
    </script>
@endpush
```

2. Tambahkan Penunjuk Arah di Main View

Terkadang ada kode Javascript yang berhubungan dengan beberapa atau bahkan semua sub view. Ketika memecahnya ke masing-masing sub view tidak mungkin dilakukan atau dikhawatirkan mengurangi *readability*, maka kode tersebut bisa tetap ditulis di main view, dengan **mengeksplisitkan** *identifier* atau bahasa mudahnya: **ada penunjuk arah**.

Contoh, dengan mockup dashboard yang sama, perlu ditambahkan sebuah filter tahun yang ketika diubah akan me-refresh ketiga sub view sekaligus.

Versi kode yang manusiawi adalah seperti berikut:

```
<!-- dashboard.blade.php -->
@extends('layout')

@section('content')

    <h1>Dashboard</h1>
    <select name="tahun" id="filterTahun">
        <option value="2021">2021</option>
        <option value="2020">2020</option>
    </select>
    <section data-role="summaryCard">
        @include('_summary')
    </section>

    <section data-role="statistic">
        @include('_chart')
    </section>

    <section data-role="popularPost">
        @include('_popularPost')
    </section>
@endsection

@push('script')
    <script>
        $('#filterTahun').on('change', function(){
          $.ajax({}).done(function (data) {
            $('[data-role="summaryCard"] .card').doSomething(data);
            $('[data-role="statistic"]').doSomething(data);
            $('[data-role="popularPost"] .item').doSomething(data);
          });
        });
    </script>
@endpush
```

Dengan menambahkan `data-role="foo"` maka pembaca kode (sekali lagi, pembaca kode ya, bukan kamu si penulisnya) bisa dengan mudah menemukan korelasi antara kode Javascript dan pasangan HTML-nya, meskipun ada di *file* yang berbeda.

Jika ingin lebih sederhana lagi, kamu bisa membuat konvensi bersama tim, semua sub view harus memiliki atribut **id** yang sama dengan nama *file*. Jika seperti itu, `<section>` pembungkus di main view bisa dihilangkan.

```
<!-- dashboard.blade.php -->
@extends('layout')

@section('content')

    <h1>Dashboard</h1>
    <select name="tahun" id="filterTahun">
        <option value="2021">2021</option>
        <option value="2020">2020</option>
    </select>
    @include('_summary')
    @include('_chart')
    @include('_popularPost')
@endsection

@push('script')
    <script>
        $('#filterTahun').on('change', function(){
          $.ajax({}).done(function (data) {
            // Dimana #summary? Sesuai konvensi, mari cari sub view yang namanya
_summary.blade.php
            $('#summary .card').doSomething(data);
            $('#chart').doSomething(data);
            $('#popularPost').doSomething(data);
          });
        });
    </script>
@endpush
```

Terapkan mana yang paling cocok buatmu (dan tim).

View Share & View Composer Terlalu Magic, Hindari!

Pernahkan kamu mengalami momen dimana sedang asik *debugging*, lalu menemukan sebuah variabel, misalnya `$categories`, tetapi tidak menemukan dari mana asal variabel tersebut. Tidak ada di Controller, tidak ada juga di View.

Kode seperti itu umum dijumpai di file Blade untuk *layouting*.

```
<!-- resources/view/layout.blade.php -->
<html>
    <head>
        <title>The Boring Stack</title>
    </head>
    <body>
        <header>
            @foreach($categories as $item)
            <a href="{{ $item->permalink }}">{{ $item->title }}</a>
            @endoreach
        </header>
        {{ $slot }}
    </body>
</html>
```

Semua yang melihat file di atas tentu bertanya-tanya, dari mana asalnya variabel `$categories`. Perlu beberapa saat sebelum kamu menyadari bahwa ini adalah salah satu ***magic*** dari Laravel.

View Share & View Composers

Jika ada variabel yang dibutuhkan di semua halaman, kamu bisa melakukannya dengan dua cara.

Pertama dengan memanfaatkan `View::share`:

```
use Illuminate\Support\Facades\View;

View::share('categories', Category::all());
```

Kedua dengan memakai View Composers:

```
View::composer('layout', function ($view) {
    $view->with('categories', Category::all());
});
```

Keduanya sama, secara *magic* mendaftarkan variabel baru yang bisa diakses dari semua View. Jika bukan kamu sendiri yang menulis kode-kode di atas, besar kemungkinan akan kesulitan ketika harus melacak asal muasalnya di kemudian hari.

> Dokumentasi tentang **View Share** dan **View Composers** bisa dibaca di https://laravel.com/docs/master/views#view-composers.

Apa Alternatifnya?

Ada satu fitur di Laravel yang harusnya lebih manusiawi jika dipakai untuk mendaftarkan variabel ke View, yaitu **Service Injection**.

Sekarang mari kita implementasikan kasus `$categories` dengan Service Injection:

```
@inject('site', 'App\Services\SiteService')

<html>
    <head>
        <title>The Boring Stack</title>
    </head>
    <body>
        <header>
            @foreach($site->categories() as $item)
            <a href="{{ $item->permalink }}">{{ $item->title }}</a>
            @endoreach
        </header>
        {{ $slot }}
    </body>
</html>
```

Sekarang kodenya terlihat lebih eksplisit dan natural. Ketika melihat `$site->categories()` secara otomatis kita akan mencari `$site` di file yang sedang dibuka saat ini (`layout.blade.php`). Ketika menemukannya di baris pertama, terlihat jelas **petunjuknya** kemana `$site` ini mengarah.

Tidak perlu lagi menerka-nerka, `dd()`, atau bertanya ke programmer lain. Petunjuknya sudah sangat jelas.

> Dokumentasi resmi tentang **Service Injection** bisa dibaca di https://laravel.com/docs/master/blade#service-injection.

Pilihan lain yang lebih tepat adalah membuat **Blade Component** khusus untuk me-render kategori.

```blade
<!-- resources/views/layout.blade.php -->
<!-- Lihat betapa bersihnya kode HTML jika memakai Blade Component       -->
<html>
    <head>
        <title>The Boring Stack</title>
    </head>
    <body>
        <header>
            <x-categories />
        </header>
        {{ $slot }}
    </body>
</html>
```

```php
<?php

// app/View/Components/Categories.php

namespace App\View\Components;

use Illuminate\View\Component;

class Categories extends Component
{
    public function render()
    {
        return view('components.categories');
    }
}
```

Yang perlu menjadi perhatian, konvensi penamaan komponen harus konsisten dan mengikuti standard Laravel.

> Dokumentasi resmi tentang **Blade Component** bisa dibaca di https://laravel.com/docs/master/blade#components.

Keuntungan

Apa yang dituliskan secara eksplisit biasanya lebih mudah dibaca, dipahami, dan diikuti alurnya. Oleh sebab itu, eksplisitkanlah pemanggilan variabel global di View dengan menggunakan **Service Injection** atau **Blade Component**.

Beberapa manfaat yang bisa kita dapat ketika menerapkannya antara lain:

1. Memaksa programmer membuat Class khusus untuk membungkus *logic* mendapatkan variabel. Disini, kita sekaligus belajar menerapkan Single Responsibility Principle.
2. Karena *logic* ada di sebuah Class, maka menjadi lebih mudah dites, dibandingkan jika logic tersebut ada di Service Provider.

> ***Single Responsibility Principle (SRP)*** adalah salah satu kaidah menulis *clean code* dimana sebuah Class harus fokus dengan satu tugas khusus, tidak boleh terlalu kompleks atau multi fungsi. **S**RP merupakan huruf pertama **(S)** dari akronim **SOLID** yang sangat tersohor itu.

Boleh, Asalkan...

1. Didokumentasikan Secara Eksplisit

Biasakan mengomentasi bagian kode yang "*magic*" untuk membantu programmer lain (atau dirimu sendiri, 3 bulan kemudian) ketika membacanya:

```
<!-- layout.blade.php -->

<!-- @kategori berasal dari ViewComposerServiceProvider -->
@foreach($kategori as $item)
...
@endforeach
```

> Ikatlah ilmu (*knowledge*) dengan mencatatnya, termasuk dengan menulis komentar yang tepat. **Bukankah manusia tempatnya lupa?**

2. Sudah Ada Konvensi

Sudah ada kesepakatan antar anggota tim yang diambil sebelumnya, bahwa semua variabel global yang ditemukan di View pasti berasal dari `ViewServiceProvider`. Tapi ingat, konvensi tanpa dokumentasi juga rawan dilupakan. Oleh sebab itu, tulislah semua konvensi di `readme.md`.

> File *readme.md* harusnya bisa menjadi sumber utama *knowledge* terkait *source code* aplikasi. Oleh sebab itu, rajin-rajinlah mencatat di `readme.md`, atau istilah kerennya "*readme driven development*".

Sub View vs Blade Component

Semua cara yang telah disebutkan sebelumnya merupakan cara singkat dan praktis untuk menjaga agar tidak ada penumpukan kode di satu *file*. Level selanjutnya, kamu bisa membuat sub view tersebut *reusable*, bisa digunakan di tempat lain, oleh programmer lain, dengan mudah. Caranya adalah dengan memanfaatkan **Blade Component**.

Secara singkat, perbedaan Sub View dan Blade Component bisa dilihat dalam tabel berikut:

SUB VIEW	BLADE COMPONENT
Mudah diterapkan karena hanya berurusan dengan file Blade	Sedikit lebih kompleks, karena harus membuat class PHP
Tidak perlu memikirkan *abstraction*	Merupakan salah satu bentuk penerapan *abstraction*
Ketika membuat sub view, kita tidak perlu berpikir *reusable*	Didesain untuk *reusable* , bisa dipakai di view lain dengan mudah
Analoginya seperti *protected method* , hanya digunakan untuk *scope* tertentu	Seperti *public method* , penerapannya lebih luas dan generik
Tidak perlu memikirkan passing parameter karena sub view otomatis bisa mengenali variable dari *parent-view-*	Perlu meng-handle parameter

Terkadang, pilihannya bukan mana yang benar atau salah, tapi **mana yang lebih cocok** dengan kondisi tim.

Dalam dokumentasi resmi Laravel terkait Blade Component, https://laravel.com/docs/master/blade#components, disebutkan: " *Components and slots provide similar benefits to sections, layouts, and includes; however, some may find the **mental model** of components and slots easier to understand*".

Kata kuncinya adalah **mental model**. Bagaimana kita mau memodelkan aplikasi. Bagaimana kita menerjemahkan kebutuhan bisnis menjadi struktur kode yang *long lasting* dan tetap mudah di-*maintain*, tiga bulan lagi, 6 bulan lagi, bahkan bertahun-tahun

dari sekarang.

Intermeso: Kode Yang Manusiawi

Sejauah ini sudah ada tiga prinsip penting yang kita ketahui untuk bisa menulis kode yang rapi dan manusiawi:

1. Memecah yang besar menjadi kecil.
2. Dekatkan yang saling membutuhkan atau yang sejenis
3. Beri petunjuk untuk hal *magic*.

Prinsip diatas bisa diterapkan untuk semua bahasa pemrograman dan *framework*.

Controller

Controller merupakan "tempat nongkrong" utama bagi programmer Laravel. Sebagian besar waktu koding akan dihabiskan disini.

- Menyiapkan data untuk View.
- Memvalidasi form.
- Memproses submit form.
- Memanggil query builder.
- Memanggil Eloquent Model.
- Pengecekan hak akses.
- Ekspor ke PDF.
- Ekspor ke Excel.
- *Looping*, *if else*, dan *logic* aplikasi akan sangat banyak ditemui di Controller.

Ketika kamu baru mengenal M-V-C, wajar jika pilihannya hanya terbatas pada ketiga komponen tersebut. Model untuk "simpan-pinjam" data dari database, View untuk menampilkan data, dan **Controller untuk sisanya**.

Tidak ada masalah untuk aplikasi skala kecil. Berpotensi obesitas (*Fat Controller*) untuk aplikasi skala menengah hingga besar.

Pada bab ini, kita akan belajar **hal-hal kecil dan mudah** yang bisa dilakukan untuk membuat Controller tetap *slim* dan mudah dipahami.

Jangan Hanya Menulis Kode, Rangkailah Sebuah Cerita!

Menulis kode untuk mesin (*compiler, interpreter*) relatif mudah. Mesin tidak punya emosi. Mesin tidak pernah lupa. Apa yang benar menurut mesin di hari ini, tetap akan dianggap benar tiga bulan kemudian. Mesin punya **konsistensi yang tinggi** dalam membaca dan menerjemahkan kode.

Sebaliknya, manusia punya perasaan dan ingatan yang terbatas. Apa yang menurut kita baik-baik saja di hari ini, bisa jadi berubah menjadi ***tragedi*** tiga bulan kemudian.

Pernah mengalami momen dimana kamu mengumpat di dalam hati, #%^@$3M!**#&, ketika membaca sebuah kode?

Ini apa ya maksudnya?

Variabel ini dari mana asalnya?

Duh, muter-muter kodenya, capek debuggingnya!

Sejurus kemudian baru tersadar, ternyata itu kodemu sendiri. Kamu menulisnya tiga bulan yang lalu, ketika masih ***fresh*** dengan masalah yang dihadapi.

Sekarang, kamu sudah lupa.

Membaca kode hanya memberikan potongan-potongan abstrak. C A minor D minor ke G ke C lagi, lupa-lupa ingat alurnya. *Debugging* berasa seperti main Minesweeper. Kodemu tidak mau berterus terang, tidak mau bercerita.

Bukan salah kode.

Tiga bulan yang lalu kamu memang hanya menulis potongan kode, **bukan cerita**.

Buat Outline (Daftar Isi)

Seberapa sering kamu corat-coret (outlining) sebelum koding? Outline merupakan salah satu metode agar kodingan lebih runut dan terstruktur.

Sebagai contoh, kita akan membuat sebuah fungsi untuk menyimpan informasi buku dan gambar *cover*-nya. Maka mulailah dengan menuliskan langkah apa saja yang diperlukan untuk menyelesaikan fungsi tersebut.

```
public function store(Request $request)
{
  // 1. validasi form
  // 2. simpan file gambar cover
  // 3. simpan data buku ke DB
  // 4. redirect dan tampilkan pesan sukses
}
```

Setelah *outline* dirasa cukup, kita tinggal menuliskan kode untuk setiap langkah. **Kerjakan dulu langkah yang paling gampang**. Jika ada kesulitan di suatu langkah, cukup tuliskan `//TODO` dan gunakan ***dummy variable*** dulu.

```php
// Kode diambil dari https://github.com/febrihidayan/laravel-blog dengan modifikasi.

public function store(Request $request)
{
  // 1. validasi form
    $this->validate($request, [
    'judul' => 'required|string|max:255',
    'isbn' => 'required|string'
  ]);
  //TODO 2. simpan file gambar cover
  $cover = "-";
  // 3. simpan data buku ke DB
  Buku::create([
    'judul' => $request->get('judul'),
    'isbn' => $request->get('isbn'),
    'pengarang' => $request->get('pengarang'),
    'penerbit' => $request->get('penerbit'),
    'tahun_terbit' => $request->get('tahun_terbit'),
    'jumlah_buku' => $request->get('jumlah_buku'),
    'deskripsi' => $request->get('deskripsi'),
    'lokasi' => $request->get('lokasi'),
    'cover' => $cover
  ]);
  // 4. redirect dan tampilkan pesan sukses
  alert()->success('Berhasil.','Data telah ditambahkan!');
  return redirect()->route('buku.index');
}
```

Pada contoh di atas, menyimpan gambar hasil upload dirasa cukup sulit untuk dilakukan. Maka kita bisa mengabaikannya dulu, namun tidak sampai menjadi *blocking* karena aplikasi tetep bisa dites dan *flow* tetap berjalan dengan normal.

Bercerita Dengan Protected Method

Baris kode adalah buah pikir programmer yang menulisnya. Sama seperti cerpen yang dihasilkan seorang penulis. Sama seperti pidato atau video motivasi yang dihasilkan seorang *public speaker*.

Pemilihan kata, pemenggalan kalimat, dan intonasi menjadi penting agar pesan tersampaikan.

Dari ketiga aspek M-V-C, **biasanya** Controller menjadi tempat yang paling sering dikunjungi (untuk dibaca atau ditulis). Oleh sebab itu menjadi penting untuk membuat sebuah Controller yang bisa "bercerita", agar pengunjung (programmer setelahmu, atau kamu sendiri tiga bulan kemudian) tidak tersesat.

Melanjutkan contoh sebelumnya, jika ada satu ***logical block*** yang dirasa cukup kompleks, kita bisa memindahkannya ke fungsi tersendiri. Dalam konsep OOP, kita bisa membuat sebuah ***protected method***.

```
// Kode diambil dari https://github.com/febrihidayan/laravel-blog dengan modifikasi.

public function store(Request $request)
{
  // 1. validasi form
    $this->validate($request, [
    'judul' => 'required|string|max:255',
    'isbn' => 'required|string'
  ]);
  //TODO 2. simpan file gambar cover
  $cover = $this->uploadCover($request);

  // 3. simpan data buku ke DB
  Buku::create([
    'judul' => $request->get('judul'),
    'isbn' => $request->get('isbn'),
    'pengarang' => $request->get('pengarang'),
    'penerbit' => $request->get('penerbit'),
    'tahun_terbit' => $request->get('tahun_terbit'),
    'jumlah_buku' => $request->get('jumlah_buku'),
    'deskripsi' => $request->get('deskripsi'),
```

```
      'lokasi' => $request->get('lokasi'),
      'cover' => $cover
    ]);

    // 4. redirect dan tampilkan pesan sukses
    alert()->success('Berhasil.','Data telah ditambahkan!');

    return redirect()->route('buku.index');

}

protected function uploadCover($request)
{
    $cover = null;

    if($request->file('cover')) {
      $file = $request->file('cover');
      $dt = Carbon::now();
      $acak  = $file->getClientOriginalExtension();
      $fileName = rand(11111,99999).'-'.$dt->format('Y-m-d-H-i-s').'.'.$acak;
      $request->file('cover')->move("images/buku", $fileName);
      $cover = $fileName;
    }

    return $cover;
}
```

Jika proses validasi form dan proses menyimpan ke database dirasa cukup kompleks, kita bisa melakukan hal yang sama untuk kedua ***logical block*** tersebut.

```
public function store(Request $request)
{
    $this->validateBuku($request);

    $cover = $this->uploadCover($request);
    $buku = $this->storeBuku($request, $cover);

    alert()->success('Berhasil.','Data telah ditambahkan!');

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

Apakah menurutmu kode di atas lebih mudah dipahami? Untuk Controller yang sederhana bisa jadi tidak terlalu terlihat bedanya. Tapi percayalah bahwa kebiasaan ini akan sangat bermanfaat ketika kompleksitas kode yang kamu buat sudah semakin meningkat.

Selamat bercerita!

Reusable Dengan Protected Method

Kelebihan lain dari *protected method* adalah *reusable*, yaitu kita bisa menggunakan *method* tersebut di tempat lain yang membutuhkan. Contoh klasik yang sering dijumpai adalah kemiripan antara proses menyimpan (*store*) dan memperbarui (*update*) buku.

```
public function update(Request $request)
{
  $this->validateBuku($request);

  $cover = $this->uploadCover($request);
  $buku = $this->updateBuku($request, $cover);

  alert()->success('Berhasil.','Data telah diubah!');

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

Bisa dilihat, setelah sebelumnya kita memindahkan *logic* upload cover ke *protected method* tersendiri, maka ketika *update* kita tinggal memanggil kembali *method* tersebut. Tidak ada duplikasi kode. Ketika ada perubahan terkait penanganan *cover file*, misalnya folder penyimpanan berubah, maka yang diubah cukup satu tempat saja, yaitu di *method* `uploadCover()`.

Reusable Dengan Trait

Protected method hanya bisa dipanggil dalam sebuah `Class` yang sama. Bagaimana jika kita juga membutuhkan fungsionalitas untuk upload cover di `Class` yang lain?

Sebagai contoh, user guest juga bisa menginput data buku, hanya saja data yang diinputkan tersebut perlu diverifikasi dulu oleh admin.

```
class BukuController extends Controller
{
    public function store(Request $request)
    {
      ...
      $cover = $this->uploadCover($request);
      ...
    }
}

class PublicBukuController extends Controller
{
    public function store(Request $request)
    {
      ...
      $cover = $this->uploadCover($request);
      ...
    }
}
```

Karena sebelumnya method `uploadCover` hanya didefinisikan di `BukuController`, maka kelas `PublicBukuController` tidak bisa mengenali method tersebut. Mau tidak mau kita harus *copy paste* dulu fungsi tersebut.

Nah, ada cara lain yang lebih tepat untuk kasus seperti ini, yaitu dengan memindahkan method `uploadCover` ke sebuah Trait.

Pertama-tama, buat sebuah Trait tersendiri, misalnya `app\Http\Traits\UploadCoverTrait`.

```php
namespace App\Http\Traits;

trait UploadCoverTrait
{
}
```

Lalu pindahkan method `uploadCover` dari Controller ke Trait tersebut.

```php
namespace App\Http\Traits;

trait UploadCoverTrait
{
    protected function uploadCover($request)
    {
        $cover = null;

        if ($request->file('cover')) {
            $file = $request->file('cover');
            $dt = Carbon::now();
            $acak = $file->getClientOriginalExtension();
            $fileName = rand(11111, 99999).'-'.$dt->format('Y-m-d-H-i-s').'.'.$acak;
            $request->file('cover')->move("images/buku", $fileName);
            $cover = $fileName;
        }

        return $cover;
    }
}
```

Selanjutnya, untuk setiap Controller yang membutuhkan fungsionalitas upload cover, cukup memanggil Trait tersebut.

```
class BukuController extends Controller
{
    use UploadCoverTrait;
    public function store(Request $request)
    {
      ...
      $cover = $this->uploadCover($request);
      ...
    }
}

class PublicBukuController extends Controller
{
    use UploadCoverTrait;
    public function store(Request $request)
    {
      ...
      $cover = $this->uploadCover($request);
      ...
    }
}
```

Selamat, kamu sudah berhasil membuat sebuah Trait yang *reusable*. Konsep ini tidak hanya terbatas di Controller. Kamu bebas (dan sangat diharapkan) untuk bisa menerapkannya di persoalan yang lain.

Maksimal Tujuh Dengan Resource Controller

Bagaimana kalau saya bilang, seberapapun kompleksnya aplikasi yang kamu bangun, jumlah Action dalam suatu Controller selalu bisa dibikin agar **tidak pernah lebih dari tujuh**.

Tujuh adalah jumlah aksi maksimal yang bisa kita lakukan terhadap suatu *resource*, paling tidak demikianlah Laravel mengajarkan kita.

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Terlebih lagi jika aplikasi yang sedang dikembangkan bertipikal CRUD, aturan **masksimal tujuh** harusnya bisa dengan mudah diterapkan. Kita tidak perlu membuat Custom Action di Controller.

Apa Itu Resource Controller?

Resource Controller adalah sebuah konsep untuk menunjukkan hubungan antara data dan aksi apa saja yang bisa dilakukan terhadap data tersebut. *Resource* biasanya mengacu ke sebuah tabel *database*, gabungan beberapa tabel (join), sub tabel (tabel dengan kondisi tertentu), kolom (atribut), atau entitas lain sesuai kebutuhan aplikasi.

RESOURCE (DATA)	CONTROLLER	CONTOH AKSI
Satu tabel	PostController	index (tampilkan semua post) store (menyimpan Post baru) destroy (hapus permanen sebuah Post)
Banyak tabel	StatisticController	index
Sub tabel	PublishedPostController	store (publish post) destroy (unpublish post)
Kolom tertentu	PasswordController	edit update ~~destroy~~ (password tidak bisa didelete)
Entitas lain	DbBackupController	index (tampilkan semua backup) store (menambah backup baru) destroy (hapus salah satu backup)

Berpikir Resource

Sampai di sini kamu sudah mengenal apa itu Custom Action dan apa itu Resource Controller, dan *goal* yang ingin dicapai adalah bagaimana menghilangkan Custom Action agar semua Controller bisa ***strict*** hanya memakai **tujuh kata**.

Apa Itu Custom Action?

Custom Action adalah ketika kamu mendefinisikan route dan method baru di Controller, di luar tujuh *method* standard.

```php
// routes/web.php
Route::resource('users', UserController::class);
Route::get('/users/downloadPdf', [UserController::class, 'downloadPdf']);

// UserController.php
class UserController extends Controller {
  public function index(){...}
  public function show(){...}
  public function create(){...}
  public function store(){...}
  public function edit(){...}
  public function update(){...}
  public function delete(){...}

  public function downloadPdf()
  {
    // generate PDF
  }
}
```

Pada contoh di atas, `downloadPdf()` merupakan Custom Action.

Studi Kasus

Mari kita latihan membuat Resource Controller dari beberapa contoh kasus yang sering kita temui.

Follow Unfollow

```
// Custom Action
UserController@follow
UserController@unfollow

// Resource Controller
FollowController@store // follow action
FollowController@destroy // unfollow action

// Pada kasus ini, kita mengganggap "Follow" adalah sebuah resource yang bisa
ditambah
// (ketika user mem-follow user yang lain) atau bisa dihapus (ketika user meng-
unfollow)
```

Upload Profile Picture

```
// Custom Action
UserController@uploadProfilePicture
// Resource Controller
UserProfilePicture@update
// Pola yang mirip, disini kita memecah `uploadProfilePicture` menjadi resource
tersendiri:
// ProfilePicture (atau UserProfilePicture agar lebih jelas) + method update.
```

Produk Global vs Produk Milik User

Contoh yang sering dijumpai dalam sebuah web marketplace adalah adanya dua halaman untuk menampilkan produk:

- URL `/produk` menampilkan produk dari seluruh user.
- URL `/<username>/produk` menampilkan produk hanya dari **user tertentu** saja.

Mari kita lihat bagaimana penerapan dari masing-masing cara penulisan Controller:

```
// Custom Action
ProductController@index
ProductController@indexToko
// Resource Controller
ProductController@index

User\ProductController@index // menggunakan namespace sebagai pembeda
//atau
UserProductController@index // menggunakan prefix sebagai pembeda
```

Referensi

- Cruddy By Design - Youtube

Single Action Controller Untuk "Sisanya"

Ketika ada kesulitan mendesain sebuah resource controller agar tetap patuh dengan **tujuh kata **, maka kita bisa memanfaatkan single action controller. Biasanya hal ini dijumpai ketika ada aksi-aksi diluar CRUD yang memang perlu ditambahkan ke dalam aplikasi.

Sebagai contoh, kita sudah mendefinisikan resource controller untuk melakukan manajemen produk:

```
Route::resource('products', ProductController::class);
```

Lalu ada kebutuhan untuk menambahkan fitur ekspor produk ke PDF, maka kita bisa membuat sebuah single action controller:

```
// routes/web.php
Route::resource('products', ProductController::class);
Route::post('/products/pdf', Products/DownloadPdf::class);

// Controller/Product/DownloadPdf.php
class DownloadPdf extends Controller
{
  public function __invoke()
  {
    // generate PDF
  }
}
```

Biasanya, aksi-aksi dalam sebuah *single action controller* tidak membutuhkan sebuah view, melainkan hanya respon dari sebuah tombol.

Beberapa contoh aksi yang cocok dijadikan *single action controller* antara lain:

- Tombol download pdf atau excel.
- Tombol logout.
- Tombol refresh cache.

- Tombol impersonate user.
- Tombol untuk men-trigger backup DB.

Referensi

- https://laravel.com/docs/8.x/controllers#single-action-controllers

Form Request Yang Terabaikan

Di bawah ini adalah tipikal kode yang sering kita jumpai. Sembilan baris kode untuk melakukan validasi form. Tentu bisa lebih jika *field* yang harus divalidasi semakin banyak.

```
class PostController extends Controller
{
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Store the blog post...
    }
}
```

Sebagai pembaca kode, mungkin kita harus scroll bolak-balik dulu sampai menemukan fungsi utama dari method `store` di atas. Validasi juga penting, tapi bukan yang utama. Ketika porsi penulisannya menjadi dominan, maka ada baiknya jika validasi form ini kita pindahkan ke tempat yang semestinya.

Biasakan membuat satu `Form Request` untuk setiap form. Kita bisa memanfaatkan Artisan CLI:

```
php artisan make:request Post/StoreRequest
```

Lalu memindahkan kode terkait validasi dari Controller ke class Request yang baru:

```php
<?php

namespace App\Http\Requests\Post;

use Illuminate\Foundation\Http\FormRequest;

class StoreRequest extends FormRequest
{
    public function rules()
    {
        return [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ];
    }
}
```

Langkah terakhir, mengubah method `store` di Controller agar *typehint*-nya merujuk ke class `StoreRequest`:

```php
use App\Http\Requests\Post\StoreRequest;

class PostController extends Controller
{
    public function store(StoreRequest $request)
    {
        \App\Models\Post::create($request->validated());
        return redirect()->back()->with('success', 'Post saved');
    }
}
```

Method Controller kita sekarang menjadi lebih langsing dan ***to the point***, dengan tidak ada fungsionalitas yang dikurangi. Validasi form tetap jalan, hanya kita pindahkan saja kodenya.

Selain untuk validasi, class Form Request juga memiliki kegunaan yang lain, antara lain:

1. Melakukan authorization atau pengecekan hak akses.
2. Memodifikasi inputan form

3. Kustomisasi pesan error

Jangan Percaya User!

Dua tempat utama dimana User bisa berinteraksi dengan aplikasi adalah:

1. Input form.
2. URL, baik hasil dari mengeklik sebuah menu atau mengetik secara manual di *address bar*.

Umumnya validasi hanya dilakukan untuk form. Kita terbiasa menambahkan "required" untuk field yang tidak boleh null ketika akan disimpan di database.

Sementara untuk URL, karena biasanya didapat dari sebuah link yang di-*generate* oleh aplikasi, seringkali kita asumsikan selalu valid. Padahal, sama seperti inputan pada form, URL yang tertulis di *address bar* juga bisa diubah oleh User.

Kita ambil contoh URL dari sebuah halaman detil artikel:

```
http://localhost/post/8
```

Dengan URL di atas, maka tipikal kode yang sering ditemui di Controller adalah seperti ini:

```
class PostController extends Controller
{
    public function show($id)
    {
        $post = Post::find($id);
        $comments = $post->comments;

        return view('post.show', compact('post', 'comments'));
    }
}
```

Sekali lagi, URL adalah kekuasaan User. Artinya, User bisa mengganti URL sesukanya (baik disengaja atau tidak). Pastikan ketika User mengetik URL yang aneh dan tidak

masuk akal, aplikasi yang kita bikin tetap mampu menanganinya dan bukan menampilkan error 500. Idealnya, aplikasi menampilkan halaman **404 not found**, atau redirect ke halaman lain.

Coba cek kembali aplikasi yang sudah pernah Anda bikin. Buka sebuah halaman yang menampilkan *list of data*, lalu klik detail. Biasanya kita akan dibawa ke halaman baru dengan format URL yang mengandung parameter ID atau slug seperti di bawah ini:

```
http://localhost/post/99990
```

```
http://localhost/post/hello-world
```

Atau kalau ingin melihat contoh nyata, silakan buka URL https://inaproc.id/berita, lalu pilih salah satu berita. Setelah itu, coba sedikit iseng dengan mengubah URL-nya, misalnya:

```
https://inaproc.id/berita/Aplikasi/Tingkatkan-Keamanan-Akun-Pengguna,-SPSE-Kini-ada-Fitur-Password-Meter
```

Diubah menjadi:

```
https://inaproc.id/berita/Aplikasi/1234
```

Apa yang ditampilkan oleh aplikasi? Halaman error, halaman 404, atau redirect ke halaman lain dengan pesan yang manusiawi ?

Disclaimer

Saya ikut andil dalam pengembangan awal aplikasi inaproc.id di atas, 4 atau 5 tahun yang lalu. Ya, saya merasa 2M (muak dan malu) dengan kodingan saya sendiri. Konon katanya itu adalah salah satu ciri seorang programmer masih belajar dan berkembang :)

Coba cek kembali kodingan kita: 3 bulan yang lalu, setahun yang lalu, atau bahkan 3 tahun yang lalu. Jika merasa baik-baik saja, mungkin "tempat main" kita "kurang jauh".

Defensive Programming

Kembali ke contoh URL berita sebelumnya, cara paling sederhana adalah menambahkan pengecekan sesaat setelah kita mendapatkan object `$post`.

```
class PostController extends Controller
{
    public function show($id)
    {
        $post = Post::find($id);
        if ($post === null) {
            abort(404);
        }
        $comments = $post->comments;

        return view('post.show', compact('post', 'comments'));
    }
}
```

Berdasar pengalaman saya sejauh ini, salah satu error yang paling sering dijumpai oleh programmer adalah `Null Pointer Exception`. Setiap kali memanggil method, kita harus memastikan return value dari method tersebut. Jika mungkin mengembalikan `null`, maka mau tidak mau kita harus menambahkan pengecekan `if null` seperti di atas.

findOrFail

Versi lebih singkat, kita bisa memanfaatkan method `findOrFail` atau `firstOrFail` bawaan Laravel:

```
class PostController extends Controller
{
    public function show($id)
    {
        $post = Post::findOrFail($id);
        $comments = $post->comments;

        return view('post.show', compact('post', 'comments'));
    }
}
```

Ketika tidak ada model Post dengan ID yang dimaksud, maka Laravel secara otomatis akan menampilkan halaman **404 not found**. Lebih sederhana dan cukup untuk kasus umum sehari-hari.

Referensi

https://laravel.com/docs/8.x/eloquent#not-found-exceptions

Hadirkan Model Secara Otomatis: Route Model Binding

Memanggil `findOrFail` bisa mengurangi *"noise"* dan melangsingkan Controller. Selanjutnya, jika ingin diet yang lebih brutal lagi, kita bahkan tidak perlu memanggil `findOrFail` sama sekali.

Sekali lagi, Laravel memiliki cukup banyak "magic" untuk mengubah ID dari URL menjadi sebuah object Model secara otomatis, tanpa kita perlu melakukan *query* secara manual.

Sebelum:

```
// routes/web.php
Route::get('post/{id}', [PostController::class, 'show']);

// PostController
public function show($id)
{
    $post = Post::findOrFail($id);

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

Sesudah:

```
// routes/web.php
Route::get('post/{post}', [PostController::class, 'show']);

// PostController
public function show(\App\Models\Post $post)
{
    return view('post.show', compact('post'));
}
```

Dengan mengubah penamaan parameter route dari `{id}` menjadi `{post}` dan menambahkan *type hint* variable `$post` di Controller, maka Laravel akan secara otomatis melakukan query untuk mendapatkan model `Post` sesuai parameter URL saat ini.

Bagi kamu yang terbiasa dengan ***strict typing*** dan bisa berdamai dengan *magic* dari Laravel, metode penulisan di atas tentu jauh lebih ringkas dan membuat Controller lebih *straightforward*.

Referensi

https://laravel.com/docs/8.x/routing#route-model-binding

Proteksi Dari Pintu Masuk Pertama: Route Constraints

Melanjutkan bahasan tentang ***strict typing*** atau ***strong typed***, maka masih ada celah dari contoh-contoh kode sebelumnya terkait parameter URL.

Kita berasumsi bahwa ID dari sebuah artikel (Post) adalah integer, karena kolom `id` di tabel `posts` tipenya juga *auto increment integer*. Lalu apa yang terjadi jika kita mengubah parameter URL menjadi bukan integer?

Contoh:

```
localhost/post/xxx
```

Memakai MySQL atau MariaDB biasanya aman-aman saja. Tetapi jika dijalankan dengan PostgreSQL akan menimbulkan error:

```
Illuminate\Database\QueryException
SQLSTATE[22P02]: Invalid text representation: 7 ERROR: invalid input syntax for
integer: "abc" (SQL: select * from "posts" where "id" = abc limit 1)
```

Error terjadi karena PostgreSQL menerapkan ***strong typed***. Kolom `id` bertipe integer, maka ekspektasinya parameter yang diberikan juga integer.

(Bukan) Solusi: Menerapkan Typehint di Controller

Ketika pertama kali menemukan kasus seperti ini, solusi pertama kali yang saya pikirkan adalah mengabaikan *Route Model Binding* dan kembali memanggil `findOrFail` secara manual. Untuk proteksinya, saya tambahkan *type hint* `int` ke parameter $id:

```php
// routes/web.php
Route::get('post/{id}', [PostController::class, 'show']);

// PostController
public function show(int $id)
{
    $post = Post::findOrFail($id);

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

Ternyata tidak berhasil.

Yang ada justru *another error*:

```
TypeError
App\Http\Controllers\PostController::show(): Argument #2 ($id) must be of type int,
string given, called in /Volumes/Code/laravolt-
sandbox/vendor/laravel/framework/src/Illuminate/Routing/Controller.php on line 54
```

PHP berharap *method* `show` dipanggil dengan parameter `int`, tapi ketika kita menemui url `localhost/post/xxx`, maka "xxx" tetap dianggap sebagai ***string***.

Solusi 2: Menerapkan Route Constraint

Solusi yang lebih baik dan elegan adalah dengan melaukan proteksi sedini mungkin. Kita sudah paham bahwa ***routes*** merupakan pintu masuk utama ke aplikasi. Semua URL yang bisa diakses oleh User wajib didefinisikan terlebih dahulu di ***routes***.

Ketika kita mendefinisian routes:

```php
Route::get('post/{id}', [PostController::class, 'show']);
```

Maka sebenarnya kita sedang bilang ke Laravel, tolong semua pola URL di bawah ini diteruskan ke `PostController@show`:

```
localhost/post/1
localhost/post/2
localhost/post/999
localhost/post/foo
localhost/post/bar
```

Programmer punya kesadaran penuh bahwa `{id}` ini adalah sebuah angka (integer). Tetapi kalau kesadaran ini tidak dituangkan dalam bentuk kode, tidak dieksplisitkan, maka Laravel tidak akan pernah tahu. Buat Laravel, semua tulisan setelah `/post/` adalah `{id}`, tidak peduli *integer* atau bukan.

Kabar baiknya, Laravel sudah menyediakan mekanisme yang sangat baik untuk menangani permasalahan di atas.

Cukup tambahkan definisi `{id}` secara eksplisit ketika kita mendefinisian *routes*:

```
Route::get('post/{id}', [PostController::class, 'show'])->where('id', '[0-9]+');

// atau memakai method alias yang lebih manusiawi,
// jika kamu tidak familiar dengan regular expression
Route::get('post/{id}', [PostController::class, 'show'])->whereNumber('id');
```

Dengan penambahan ini, maka pengecekan tipe atau pola parameter URL akan dilakukan di level *routes*. Controller cukup terima bersih parameter yang sudah valid tipe atau polanya.

Referensi

https://laravel.com/docs/8.x/routing#parameters-regular-expression-constraints

Intermeso: Happy Case, Alternative Case, Edge Case

Kasus terakhir adalah contoh dimana kita sebagai programmer hanya fokus ke *happy case* atau kasus ideal. Kasus dimana datanya valid, lingkungan atau resource mendukung, dan tidak ada orang iseng diluar sana.

Dunia tidak seindah itu.

Pada prakteknya, banyak *edge case* dan *alternative case* yang harus di-*handle*, dimana hal tersebut juga merupakan tanggung jawab programmer. Jangan hanya bergantung ke Analis untuk mendefinisikan semua kemungkinan buruk atau bergantung sepenuhnya ke Tester untuk mencari *bug* dari fitur yang kita koding. Mulai sekarang biasakan untuk sedikit "paranoid" sekaligus menanamkan *malicious mindset*.

Bagaimana jika inputnya asal?

Bagaimana jika ID yang dicari tidak ada di *database*?

Bagaimana jika struktur *array* tidak seperti yang kita harapkan?

Bagaimana jika format sebuah string (misal JSON) tidak valid?

Bagaimana jika filenya tidak terbaca?

Bagaimana jika koneksi API putus di tengah proses?

Bagaimana jika, apapun penyebabnya, proses menyimpan ke database gagal?

Pada awalnya, akan banyak validasi (dan `if`) yang perlu ditambahkan agar kode kita menjadi lebih *defensive*. Ujung-ujungnya membuat baris kode menjadi membengkak lagi. Seiring waktu dan pengalaman, kita akan menemukan banyak cara lain untuk menghapus `if`, misalnya dengan penerapan *typehint*, Data Transfer Object, atau Null Object.

Model

Jika Controller menjadi tempat nongkrong, tempat yang paling sering disinggahi untuk ditulisi, maka biasanya Model menjadi *object* yang paling sering dipanggil. Ya, karena fungsinya sebagai "perantara" antara aplikasi dengan database, tidak heran jika pemanggilan Model bakal ditemui disana-sini.

Oleh sebab itu, menjadi penting untuk "memelihara" Model agar tetap terlihat "ramping" dan mudah dipahami. Pada bab ini, kita akan melihat beberapa contohnya.

Tetap Rapi Dengan Trait

Ketika membahas Controller pada bab sebelumnya, kita sudah memanfaatkan Trait untuk menghindari duplikasi atau biasa disebut dengan *code reuse*. Pada prakteknya Trait juga bisa digunakan untuk keperluan yang lain, salah satunya adalah mengelompokkkan *method* yang sejenis.

Contoh, ketika membuat Eloquent Model, beberapa jenis *method* yang sering dibuat adalah terkait *relationship*, *setter getter*, dan *scope*.

Mari kita lihat contoh kode berikut:

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

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

    public function getFullnameAttribute()
    {
        return $this->firstname.' '.$this->lastname;
    }

    public function addresses()
    {
        return $this->hasMany(Address::class);
    }

    public function scopeActive($query)
    {
        return $query->where('status', 'ACTIVE');
    }

    public function setFirstnameAttribute($name)
    {
        $this->attributes['firstname'] = ucfirst($name);
    }

    public function scopeIdle($query)
    {
        return $query->whereDate('last_login', now()->subMonth(1));
    }
}
```

Jika kita kelompokkan berdasar jenisnya, maka potongan kode di atas memiliki pola:

- relationship
- relationship
- accessor

- relationship
- scope
- mutator
- scope

Sejatinya tidak ada yang salah dengan cara penulisan di atas, hanya saja terlihat "berserakan".

Sesuai prinsip pada bab 1 **"dekatkan yang sejenis"**, bukankah lebih bagus jika kita bisa mengelompokkan *method* sesuai jenisnya? *Relationship* berdekatan dengan *relationship*, *scope* berdekatan dengan *scope*, dan seterusnya. Dari contoh di atas, maka urutan penulisan yang lebih baik adalah:

- relationship
- relationship
- relationship
- scope
- scope
- accessor
- mutator

Nah, Trait bisa dimanfaatkan untuk **"memaksa"** programmer agar mengikuti aturan penulisan di atas.

```php
namespace App\Models;

class User extends Model
{
    use \App\Models\Traits\UserRelationship;
    use \App\Models\Traits\UserScope;
    use \App\Models\Traits\UserAccessor;
    use \App\Models\Traits\UserMutator;
    // method lain yang tidak masuk kategori apapun
}
```

Selanjutnya, kita cukup membuat Trait baru untuk masing-masing jenis *method*.

```php
// app/Models/Traits/UserRelationship.php

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

```php
// app/Models/Traits/UserScope.php

trait UserScope
{
    public function scopeIdle($query)
    {
        return $query->whereDate('last_login', now()->subMonth(1));
    }
    ...
}
```

```php
// app/Models/Traits/UserAccessor.php

trait UserAccessor
{
    public function getFullnameAttribute()
    {
        return $this->firstname.' '.$this->lastname;
    }
    ...
}
```

```php
// app/Models/Traits/UserMutator.php

trait UserMutator
{
    public function setFirstnameAttribute($name)
    {
        $this->attributes['firstname'] = ucfirst($name);
    }
    ...
}
```

Ketika nanti ada programmer baru hendak menambahkan sebuah *method* terkait *relationship*, dia akan membaca file `User.php` dan melihat sebuah petunjuk yang jelas: **"Hai, ingin menambahkan relationship baru? Ini lho sudah kami siapkan tempatnya di Trait UserRelationship"**.

Tentu tetap butuh sosialisasi dan kesadaran bersama agar aturan penulisan di atas bisa diikuti, terutama bagi programmer pemula. Tapi minimal kita sudah menyiapkan kerangka kode yang lebih mudah diikuti.

Kebiasaan baik tidak cukup hanya dengan niat. Perlu **"fasilitas"** agar kebiasaan tersebut juga mudah dilakukan. Trait adalah salah satu fasilitas yang bisa kita manfaatkan untuk membuat organisasi kode yang lebih manusiawi.

Di folder mana Trait dibuat dan bagaimana penamaannya bisa disesuaikan dengan karakteristik proyek. Contoh yang kami berikan mungkin cocok untuk proyek skala kecil hingga menengah. Untuk skala yang lebih besar, kita bisa sesuaikan lagi struktur foldernya, misal menjadi seperti ini:

```
└── app
    └── Models
        ├── Post
        │   ├── PostModel.php
        │   ├── PostRelationship.php
        │   └── PostScope.php
        └── User
            ├── UserAccessor.php
            ├── UserModel.php
            ├── UserMutator.php
            ├── UserRelationship.php
            └── UserScope.php
```

Semakin sering mengerjakan proyek, maka kita akan semakin pintar **mengenali pola**. Temukan mana pola yang cocok buat kamu dan tim.

Encapsulation Dengan Accessor

Enkapsulasi adalah salah prinsip dalam *Object Oriented Programming* dimana kita dilarang untuk mengakses secara langsung *property* sebuah *object*. Jika butuh sesuatu dari *object* tersebut, silakan panggil *method* yang sudah disediakan.

```
public function getFullnameAttribute()
{
    return $this->firstname.' '.$this->lastname;
}
```

Method `getFullnameAttribute` di atas adalah contoh enkapsulasi.

Alih-alih memanggil kode berikut di setiap file Blade:

```
<div>{{ $user>firstname.' '.$user->lastname }}</div>
```

Kita cukup memanggil:

```
<div>{{ $user->fullname }}</div>
```

Apa tujuannya?

Agar ketika ada perubahan implementasi, si pemanggil *method* (atau biasa disebut *Client Class*) tidak ikut berubah. Jadi, selama *public method* nya masih sama, semua file atau *class* yang memanggil `getFullnameAttribute()` tidak perlu tahu apakah detil implementasinya berubah atau tidak.

Sebagai contoh, selain `firstname` dan `lastname`, sekarang seorang user juga punya atribut `middlename`. Tentu saja ini akan mengubah *logic* untuk mendapatkan `fullname`. Tapi karena kita sudah menerapkan enkapsulasi, perubahan tersebut cukup dilakukan di

satu tempat saja:

```
public function getFullnameAttribute()
{
    return $this->firstname.' '.$this->middlename.' '.$this->lastname;
}
```

Semua kelas atau file Blade yang sudah terlanjur memanggil method `fullname` tidak perlu diubah.

Begitulah indahnya enkapsulasi.

Mengganti Accessor Dengan Getter

Jika kamu tipikal programmer yang tidak terlalu suka dengan "magic", maka membuat method **getter** secara tradisional masih tetap boleh dilakukan.

Getter adalah sebuah *method* dalam suatu kelas yang fungsinya adalah meng-enkapsulasi pemanggilan atribut.

```
class User extends Model {
  public function getFullname()
  {
    $sapaan = ($this->gender === 'PRIA') ? "Bapak" : "Ibu";
    return $sapaan . ' ' . $this->name;
  }
}
```

Karena *getter* adalah sebuah *method* biasa, maka kita bisa dengan mudah membedakan mana getter dan mana atribut biasa yang diambil dari kolom basis data. Tidak ada lagi keraguan ketika membaca kode di bawah ini:

```
{{ $user->name }} // "name" adalah sebuah kolom
{{ $user->getFullname() }} // getter dipanggil sebagai sebuah method
```

Standardisasi Accessor Dengan Prefix

Salah satu opsi yang bisa dilakukan adalah dengan membuat aturan penamaan accessor agar mudah dipahami oleh setiap programmer yang membacanya. Salah satu aturan yang bisa diterapkan adalah dengan memberikan prefix tertentu, misalnya `present` (dari kata Presenter).

```
class User extends Model {
  public function getPresentFullnameAttribute()
  {
    $sapaan = ($this->gender === 'PRIA') ? "Bapak" : "Ibu";
    return $sapaan . ' ' . $this->name;
  }
}
```

Dengan mengikuti standard seperti di atas, maka programmer yang membaca kode di bawah ini bisa langsung paham bahwa `present_fullname` bukanlah sebuah kolom, tapi sebuah **accessor**.

```
{{ $user->name }} // Ok, saya tahu "name" adalah sebuah kolom di database
{{ $user->email }} // begitu juga email
{{ $user->present_fullname }} // ada prefix "present", berarti ini bukan kolom,
easy...
```

Tentunya kamu bebas memberi aturan penamaan yang lain, yang penting **konsisten dan mudah dimengerti**.

Bercerita Dengan Scope

Anggaplah kita akan membuat sebuah halaman `popular-post` dengan spesifikasi:

1. *Popular post* adalah artikel dengan **jumlah view lebih dari 1000** dan **jumlah comment lebih dari 100**.
2. Hanya artikel yang **diterbitkan** dalam 2 hari terakhir yang bisa dianggap sebagai *popular post*.
3. Artikel dengan jumlah **view terbanyak ditampilkan di awal**.
4. Pengunjung bisa melakukan **pencarian berdasar judul dan isi artikel**.

Tipikal kode yang akan dibuat di Controller biasanya seperti di bawah ini:

```
class PopularPostController extends Controller
{
    public function index()
    {
        $query = Post::where('status', 'PUBLISHED')
            ->where('published_at', '>', now()->subDays(2))
            ->where('view_count', '>', 1000)
            ->where('comment_count', '>', 100);

        if ($keyword) {
            $query->where(function ($query) use ($keyword) {
                $query->where('post.title', 'like', '%'.$filter.'%')
                      ->orWhere('post.content', 'like', '%'.$filter.'%');
            });
        }
        $posts = $query->orderByDesc('view_count')->paginate();

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

Cukup familiar kan?

Ada yang salah?

Tidak ada.

Selama fiturnya berfungsi dan tidak ada bug, maka kode bisa dianggap benar. Tapi, selalu ada cara untuk memperbaiki kode, menyiapkan kode hari ini agar mudah dipahami oleh orang lain di masa yang akan datang.

Bayangkan Outline-nya

Untuk contoh kasus di atas, kita bisa memanfaatkan Query Scopes untuk memindahkan *logical block* yang berdekatan (sejenis) dari Controller ke Model.

Coba kita bandingkan kode sebelumnya dengan kode di bawah ini. Menurutmu mana yang sekilas lebih mudah dipahami?

```
class PopularPostController extends Controller
{
    public function index()
    {
        $posts = Post::query()
            ->published()
            ->popular()
            ->filterByKeyword($keyword)
            ->paginate();

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

Controller yang sebelumnya berisi barisan kode prosedural yang harus kita pahami setiap barisnya, sekarang berubah hanya berisi *outline* atau **petunjuk** saja. Dari sini, kita bisa dengan mudah melanjutkan mau "membuka" *method* yang mana. Butuh melakukan perbaikan terkait pencarian, tinggal buka *method* `scopeFilterByKeyword` di model `Post`.

Keuntungan lain, kita bisa dengan mudah menerapkan fungsionalitas yang sama di tempat lain. Butuh fitur pencarian di halaman admin? Cukup panggil `->filterByKeyword()`.

Don't Repeat Yourself!

Implementasi Scope-nya

Setelah kita bisa membuat *outline*-nya , langkah berikutnya tentu saja tinggal membuat *scope* dan memindahkan *logical block* yang sebelumnya berada di Controller.

```
class Post extends Model
{
    public function scopePopular($query)
    {
        return $query
            ->where('published_at', '>', now()->subDays(2))
            ->where('view_count', '>', 1000)
            ->where('comment_count', '>', 100)
            ->orderByDesc('view_count');
    }
    public function scopePublished($query)
    {
        return $query->where('status', 'PUBLISHED');
    }

    public function scopeFilterByKeyword($query, string $keyword)
    {
        // Validasi $keyword bisa kita pindahkan ke scope, sehingga Controller tidak
perlu ada "if" lagi
        if (!$keyword) {
            return $query;
        }
        return $query->where(function ($query) use ($keyword) {
                $query->where('post.title', 'like', '%'.$filter.'%')
                      ->orWhere('post.content', 'like', '%'.$filter.'%');
            });
    }
}
```

Cukup mudah kan?

Kita juga bisa mengoptimasi kode lebih jauh lagi, dengan memindahkan *method* terkait *scope* ke Trait tersendiri. Tapi hal tersebut opsional saja. Sesuaikan dengan

kompleksitas aplikasi yang sedang kita kembangkan. Jangan sampai melakukan *premature optimization*.

Referensi

https://laravel.com/docs/8.x/eloquent#query-scopes

Pegawais? Penggunas? Saatnya "Break The Rules!"

Kita tahu bahwa satu Eloquent Model merepresentasikan satu buah tabel di *database*. Kita juga sudah paham dengan konvensi penamaan tabel yang merupakan bentuk jamak dari nama model.

MODEL (SINGULAR)	NAMA TABEL (PLURAL)
User	users
Book	books
Box	boxes
Person	people (hmm)
Ox	oxen (???)

Dalam bahasa Inggris, perubahan dari singular menjadi bentuk plural dibagi menjadi dua:

1. Yang beraturan, biasanya ditambahi `s` atau `es` dibelakang kata.
2. Yang tidak beraturan, sesuai dua contoh terakhir di atas.

Itulah kenapa sering dijumpai nama tabel yang aneh dibaca seperti **penggunas** dan **pegawais**. Kita membuat aplikasi dengan *domain* Indonesia, tapi mengikuti konvensi bahasa Inggris. Jadinya kan aneh.

Konvensi nama Model dan nama tabel dibuat untuk memudahkan koding. Namun, ketika konvensi tersebut terasa aneh, maka kita boleh melanggarnya. **Break The Rules!**.

Cukup *override* `$table` dan kita bisa menentukan nama tabel dengan bebas, tentunya nama yang lebih manusiawi.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Pegawai extends Model
{
    protected $table = 'pegawai';
}
```

Semoga setelah membaca ini, kita dijauhkan dari memberi nama tabel yang tidak manusiawi.

> Beberapa orang menyarankan untuk *strict* memakai bahasa Inggris ketika memberi nama kelas, variabel, ataupun tabel, meskipun aplikasi yang kita buat untuk kebutuhan orang Indonesia. Menurut opini saya, penamaan yang paling tepat adalah yang konsisten dan sesuai dengan istilah yang sering digunakan sehari-hari oleh pengguna.
>
> Sebagai contoh, jika klien dari pemerintahan sudah terbiasa dengan istilah **pegawai**, maka akan lebih bijak jika istilah tersebut konsisten dipakai mulai dari dokumen analisis, penamaan dalam koding, hingga desain *database*. Tidak perlu memaksakan menjadi ***employee***.

Referensi

- https://laravel.com/docs/master/eloquent#table-names

Pasti Aman Dengan withDefault()

Seberapa sering kamu mendapatkan error **Trying to get property of non-object**?

Contoh paling klasik penyebab error tersebut skenario **User *has one* Profile**, dengan contoh *relationship* seperti di bawah ini:

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

Pada kondisi ideal, memanggil `$user->profile->bio` untuk menampilkan bio dari seorang *user* sepertinya aman-aman saja. Kita kan sudah mendefinisikan *relationship*-nya.

Namun apa jadinya jika ternyata ada sebuah data di tabel `users` yang tidak ada relasinya di tabel `profiles`? Ada banyak hal yang bisa menjadi penyebabnya:

1. Menyederhanakan proses registrasi, sehingga User tidak diminta untuk mengisi profile dari awal.
2. *Legacy data* dari aplikasi lama dimana waktu itu belum ada isian profil User secara lengkap.
3. Kesalahan koding, misalnya tidak menerapkan *database transaction*, sehingga menyebabkan rusaknya integritas data.

Pada kasus-kasus di atas, memanggil `$user->profile->bio` jelas berpotensi menimbulkan error **Trying to get property of non-object**.

Ketika kita tidak punya kuasa untuk memastikan **integritas data** di level basis data, maka mau tidak mau kita harus menerapkan kembali *defensive programming* di level kode. Dalam kasus ini, `withDefault` merupakan pilihan yang tepat.

```
class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class)->withDefault();
    }
}
```

Dengan penambahan `withDefault` seperti di atas, maka memanggil `$user->profile` dijamin tidak akan *return* `null`. Jika tidak ada data terkait User tersebut di tabel `profiles`, `$user->profile` akan tetap mengembalikan *object* `Profile`, hanya saja semua atributnya bernilai `null`.

Jika ingin mengeset *default value* dari suatu atribut, kita bisa menambahkan:

```
class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class)->withDefault(['bio' => '-masih kosong-
']);
    }
}
```

Relationship yang bisa ditambahkan `withDefault` adalah:

1. belongsTo
2. hasOne
3. hasOneThrough
4. morphOne

Dokumentasi lengkap terkait `withDefault` bisa dibaca di https://laravel.com/docs/8.x/eloquent-relationships#default-models.

PHP 8.0: Null-Safe Operator

Sejak PHP 8.0, kita bisa menghindari error "sejuta umat" tersebut dengan menerapkan operator `?->`:

```
$user->profile?->bio;
```

Kode di atas tidak akan menghasilkan *error* meskipun `$user->profile` bernilai `null`.

> Ada banyak fitur baru sejak PHP 8.0 yang membuatnya terasa lebih modern. Kita akan membahasnya lebih mendalam di buku tersendiri :)

> Null Object merupakan salah satu *design pattern* yang bisa dimanfaatkan untuk menghindari pengecekan `if($foo !== null)` di banyak tempat.

Referensi

- https://designpatternsphp.readthedocs.io/en/latest/Behavioral/NullObject/README.html
- https://docs.php.earth/php/ref/oop/design-patterns/null-object/
- https://php.watch/versions/8.0/null-safe-operator

Pasti Konsisten Dengan DB Transaction

Melanjutkan skenario User *has one* Profile sebelumya, anggaplah sekarang kita memiliki sebuah form registrasi yang cukup panjang, ala-ala website pemerintahan gitu lah ^_^.

Ketika form tersebut di-*submit*, kita perlu menyimpannya ke dalam 2 tabel sekaligus: `users` dan `profiles`. Kira-kira kodenya seperti ini:

```
class Register
{
    public function __invoke(Request $request)
    {
        $user = new User;
        $user->email = $request->email;
        $user->password = $request->password;
        $user->save();
        $profile = new Profile;
        $profile->name = $request->name;
        $profile->bio = $request->bio;
        $user->profile()->save($profile);
        // show success message
    }
}
```

Pernah ya menulis kode seperti di atas?

Kalaupun bukan User dan Profile, saya yakin kita semua sering membuat sebuah fungsi untuk melakukan beberapa operasi ke database sekaligus.

Sekarang pertanyaannya, mungkinkah data user berhasil disimpan tetapi profilnya tidak tersimpan ke database?

Jawabannya **sangat mungkin**.

Bisa jadi ada kesalahan ketika proses menyimpan profil, terutama ketika proses *development*. Meskipun hanya data development, tetap saja memiliki data yang integritasnya tidak terjamin terasa tidak menyenangkan.

Bisa jadi setelah memanggil `$user->save()`, tiba-tiba *server* down sehingga kode untuk menyimpan profil belum sempat dieksekusi. Peluangnya mungkin sangat kecil, tapi tetap saja mungkin terjadi :)

Untuk kasus seperti itu, menerapkan *database transaction* merupakan pilihan yang tepat.

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

public function __invoke(Request $request)
{
    DB::transaction(function () {
        $user = new User;
        $user->email = $request->email;
        $user->password = $request->password;
        $user->save();

        $profile = new Profile;
        $profile->name = $request->name;
        $profile->bio = $request->bio;
        $user->profile()->save($profile);

    });
}
```

Sekarang, hanya ada dua kemungkinan:

1. Keduanya berhasil, sehingga integritas data terjaga.
2. Atau, ketika salah satu gagal, maka proses yang lain otomatis akan dibatalkan. Dalam hal ini, integritas data juga tetap terjaga.

Tidak ada lagi ceritanya berhasil menyimpan ke tabel `users` tapi gagal menyimpan ke tabel `profiles`.

Referensi

- https://laravel.com/docs/8.x/database#database-transactions

Intermeso: Berdamai Dengan Eloquent Model

Ketika menyebut kata "Model" di Laravel, maka yang kita maksud adalah Eloquent Model. Eloquent sendiri merupakan sebuah Object-Relational Mapper (ORM) yang menjadi jembatan antara programmer dengan database.

Kita koding PHP dengan pendekatan OOP, sedangkan (kebanyakan) database memiliki karakteristik *relational* (familiar dengan istilah Relational Database Management System atau RDBMS?). Oleh sebab itu muncul konsep ORM, dimana kita berinteraksi dengan *relational database* melalui pendekatan *object*.

```
// OOP
$user = new User;
$user->email = 'me@theboringstack.com';
$user->save();

// query yang dihasilkan RDBMS
// insert into users (email) values ('me@theboringstack.com');
```

Eloquent sendiri menurut saya ibarat pisau bermata dua. Di satu sisi sangat memudahkan untuk berinteraksi dengan database, terutama dalam proses *insert*, *update*, *delete* atau *query* sederhana.

Namun sisi lainnya bisa "menjerumuskan" ketika kita harus melakukan *query* atau pengambilan data yang cukup kompleks. Antara kodenya jadi susah dibaca (karena query yang kompleks tersebut harus diterjemahkan menjadi kode Eloquent yang sangat panjang dan *nested*) atau performanya jadi lambat.

So, knowing your tools!

Ada saatnya kita menyerah dan tidak memaksa **"harus"** Eloquent.

Ada kalanya *plain SQL query* lebih manusiawi untuk dipahami.