Book Cover

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*.

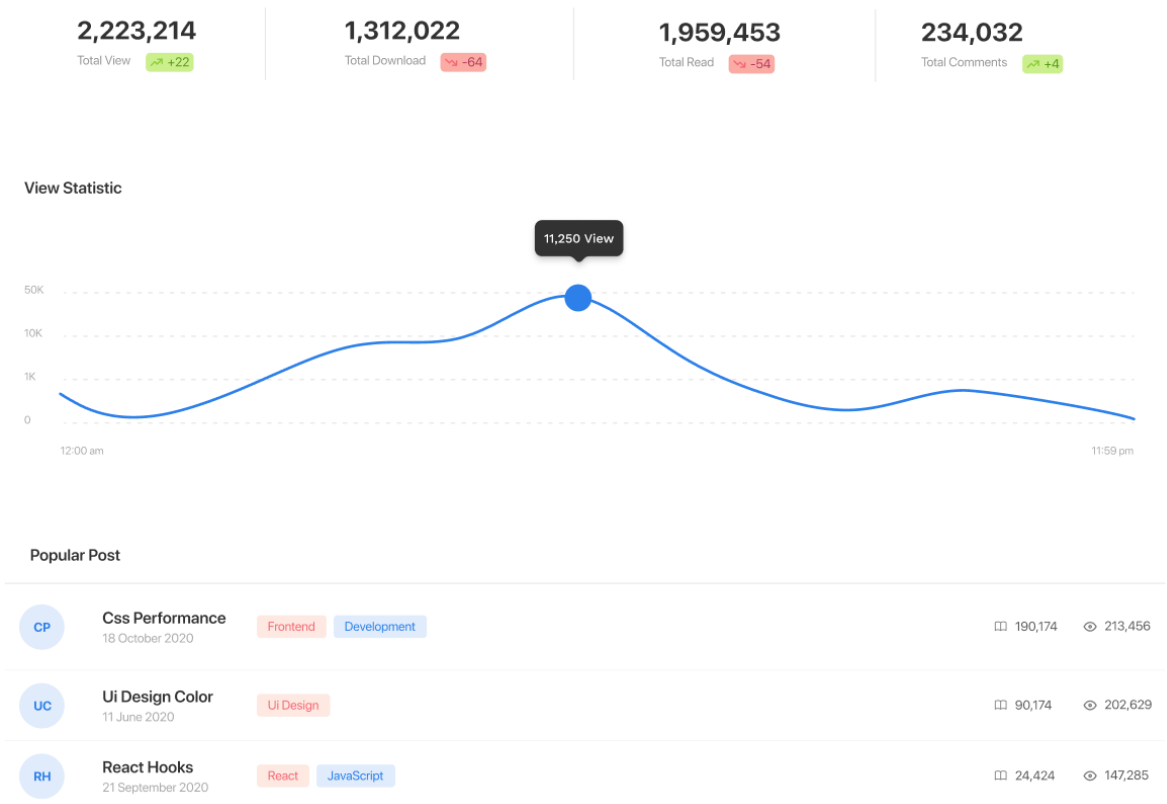
Di awal proyek, semua masih terlihat rapi. Kodenya masih sedikit. Seiring berjalannya waktu, ada penambahan fitur disana-sini, tambal sulam *bug* di kanan dan di kiri. Kode yang awalnya masih terlihat dalam satu layar sekarang harus di-*scroll* berkali-kali untuk melihat keseluruhan isinya.

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 file blade seperti berikut:

```
@extends('layout')

@section('content')
    <h1>Statistik Laporan</h1>
    <section>
        ...
        Summary
        ...
    </section>
    <section>
        ...
        Chart
        ...
    </section>
    <section>
        ...
        List
        ...
    </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

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. `_list.blade.php`

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

```
@extends('layout')

@section('content')

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

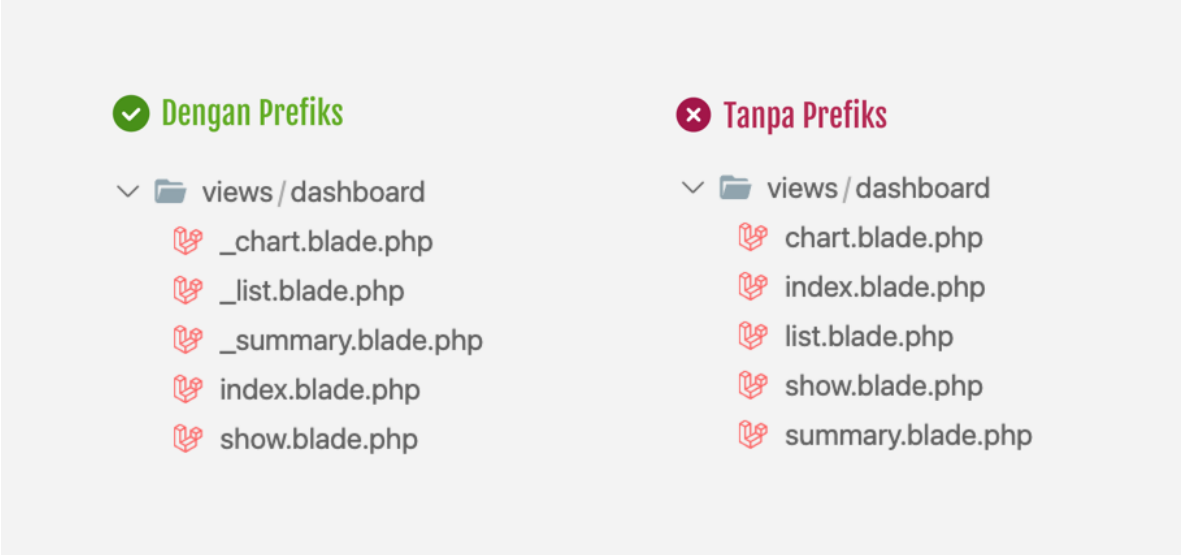
Alih-alih punya satu *file* yang berisi 100 baris kode, sekarang kamu punya 4 *file* yang masing-masing berisi 25 baris kode. Lebih rapi dan lebih ringan ketika dibuka di *code editor* atau IDE.

> 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*.

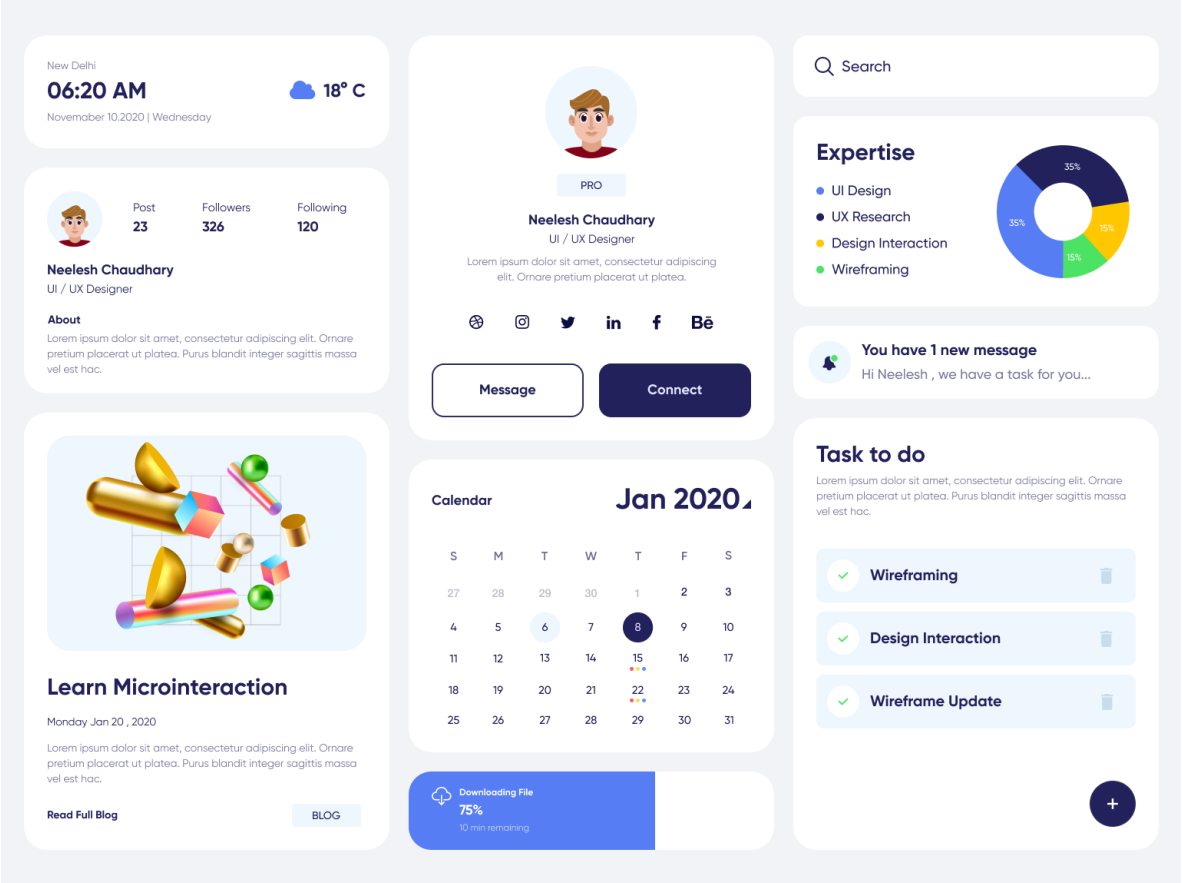
Lebih jauh lagi, kamu juga bisa membuat folder baru untuk meletakkan *sub view*. Nama yang umum dipakai biasanya **partials** atau **sub**. Kalau sudah dibuatkan folder khusus untuk menampung *sub view*, maka nama filenya tidak perlu lagi diberi prefiks "_" (*underscore*).

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 Mencampur PHP dan JS

Pemakaian editor atau IDE sudah menjadi hal yang wajar bagi programmer saat ini. Setiap keyword, setiap bahasa pemrograman bisa kamu atur *syxtax higlighting*-nya untuk memudahkan mengenali kode.

//TODO gambar syntax higlighting bisa mengenali error

Mencampur kode PHP dan Javascript (berlaku juga untuk bahasa lain) akan mengurangi readability (seberapa mudah kode dibaca/dipahami) dan kemampuan editor/IDE untuk menganalisis kode.

//TODO gambar contoh kode blade + JS campur

Ketika kebutuhan aplikasi mengharuskan adanya "interaksi" antara kode Blade(PHP) dan Javascript, misalnya variable Javascript yang berasal dari variable PHP, ada dua cara yang bisa dilakukan:

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

Passing Variable Sebagai `data-attribute`

// TODO contoh kode

Dengan metode penulisan seperti di atas, kita bisa meminimalisir adanya ***kode oplosan***, yaitu suatu kondisi bercampurnya 2 bahasa dalam **satu baris kode**.

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.

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

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 proses belajar hal baru. Hindarilah semaksimal mungkin!

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`.

//TODO contoh kode

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 sembarang tempat.

Idealisme vs kompromi

Jangan Pisahkan JS dan Pasangan HTML-nya

Kasus yang sering ditemui ketika koding di View adalah menambahkan Javascript untuk membuat halaman yang lebih interaktif.

Contoh pertama, menggunakan halaman dashboard sebelumnya, ternyata perlu ada tambahan tombol "Export Excel" di bagian Tabel.

Contoh kedua, kita mau menambahkan filter dengan mekanisme Ajax agar tidak perlu *refresh* halaman. Kira-kira alur kodenya seperti ini:

1. Tambahkan event onclick di tombol "Tampilkan"
2. Request ke server via Ajax
3. Update *chart*
4. Update tabel

Karena aksi ini melibatkan beberapa sub view, maka lebih tepat jika kode Javascriptnya diletakkan di view utama.

//TODO skeleton kode

Untuk memudahkan pembacaan kode, maka disarankan untuk menambahkan *identifier* di view utama, misalnya menggunakan atribut **"id"** yang berfungsi sebagai "rambu penunjuk arah". Jadi, ketika nanti ada programmer yang membaca kode Javascript, dia bisa langsung menentukan pasangan kode HTML-nya ada di sub view yang mana tanpa harus menelusuri satu per satu.

:bulb: Ada dua prinsip penting yang harus dibiasakan untuk bisa menulis kode yang rapi:

1. **Memecah** yang besar menjadi beberapa bagian kecil.
2. **Dekatkan** yang saling membutuhkan.

Resapi, pahami, praktekkan, dan biasakan. Prinsip diatas berlaku di semua bahasa pemrograman dan framework.

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`:

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

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

Kedua dengan memakai View Composers:

```php
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*".