

Pak Sandhika Galih - Belajar Laravel 8

https://youtube.com/playlist?list=PLFIM0718LjIWiihbBlq-SWPU6b6x21Q_2&si=lzJiu8C3YTHLEOGA

1. Intro

Apa itu laravel?

Laravel is a web application framework with expressive, elegant syntax. We've already laid the foundation – freeing you to create without sweating the small things. ~ <https://laravel.com/>

Laravel adalah kerangka aplikasi web dengan sintaksis yang ekspresif dan elegan. Kami telah meletakkan fondasinya – memberi Anda kebebasan untuk berkreasi tanpa harus memusingkan hal-hal kecil.

Web Application Framework

Requirement

Code editor : VSCode

Extension :

- PHP Intelephense
- Laravel Artisan
- Laravel Snippets (Winnie Lin)
- Laravel Blade Snippets (Winnie Lin)
- Laravel Blade Spacer (Austen cameron)
- Laravel GoTo View (codingyu)

2. Instalasi & Konfigurasi

Dokumentasi Laravel 8 : <https://laravel.com/docs/8.x>

Ada beberapa cara membuat atau menginstal project laravel.
Bisa menggunakan **Docker Compose**, **composer**, atau **laravel installer**.

```
composer create-project laravel/laravel:^8.0 example-app
cd example-app
php artisan serve
```

Laravel valet (opsional)

<https://laravel.com/docs/8.x/valet#main-content>

```
sudo nano ~/.bash_profile
```

tambahkan :

```
~/.bash_profile
export PATH=$PATH:~/composer/vendor/bin
```

```
source ~/.bash_profile
```

```
laravel
```

```
. ~/.bash_profile
```

```
laravel new coba2-laravel
```

[Belum Finish]

3. Struktur Folder, Routes & View

Dokumentasi : <https://laravel.com/docs/8.x/structure>

3 folder yg harus diingat tempat penyimpanannya :

- Models
- Views
- Controllers

Folder/file yg harus diperhatikan :

- routes : tersimpan file-file yang untuk kita melakukan routing / penjaluran.

web.php : (web routes) penjaluran untuk web pada aplikasi kita

- public : untuk menyimpan file-file static atau asset-asset kita
- .env : file environment (untuk koneksi database, dll.)
jika menggunakan valet, silakan ganti APP_URL = http://localhost
menjadi http://coba-laravel.test/

File routes/web.php

```
Route::get('/', function () {
    return view('home');
});
```

Tambahkan File resources/home.blade.php

- isikan script html yg akan ditampilkan pada home

Bikin File style.css di folder /public

bisa diletakkan di dalam folder css sendiri (/public/css/style.css)

hubungkan css pada /home.blade.php dengan menambahkan tag link langsung "style.css" atau "css/style.css" karena URL pada views sudah relative terhadap folder /public

begitu pula dengan javascript

```
<link rel="stylesheet" href="style.css">
atau
<link rel="stylesheet" href="css/style.css">

javascript
<script src="js/script.js"></script>
```

Mengirimkan data dari routes kita ke view, karena mungkin saja data tadi berubah-ubah sesuai user yg login

```
Route::get('/about', function () {
    return view('about', [
        "name" => "Akhmad Faiz Abdulloh",
        "email" => "akhmadfaizabdulloh@gmail.com",
        "image" => "cute.jpeg"
    ]);
});
```

```
<body>
    <h1>Halaman About</h1>
    <h3><?= $name; ?></h3>
    <p><?= $email; ?></p>
    " width="200">
</body>
```

[Belum Finish]

4. Blade Templating Engine

Templating engine merupakan sebuah fitur/tools untuk membantu kita dalam mengelola tampilan halaman web, khususnya untuk sebuah framework.

Dan untuk laravel, templating engine-nya dinamakan dengan Blade

Dokumentasi : <https://laravel.com/docs/8.x/blade>

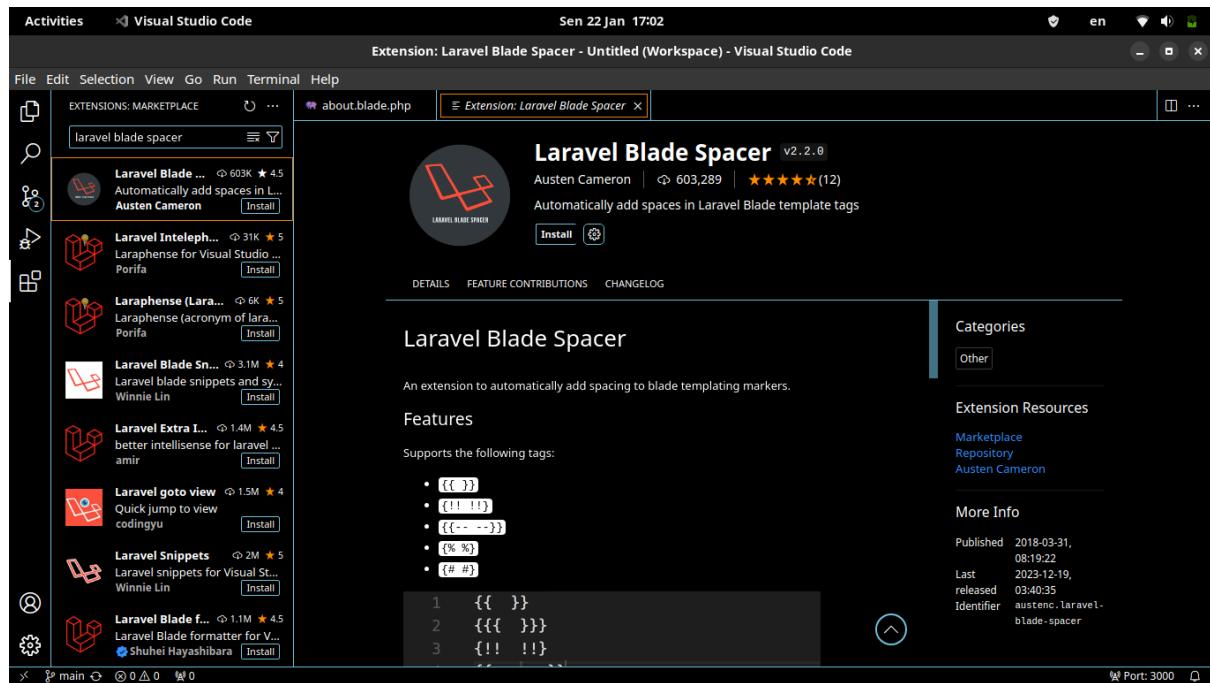
You may display the contents of the name variable like so:

Hello, {{ \$name }}.

Blade's {{ }} echo statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

- Install VS Code Extensions :

Laravel Blade Spacer (by : Austen Cameron)



`{{ $name }}` sama dengan php echo `<?= $name; ?>`

jika kita ke folder **Storage > framework > views**
maka laravel akan otomatis meng-compile script blade

Pakai Bootstrap untuk layouting-nya

<https://getbootstrap.com/docs/5.0/getting-started/introduction/>

Layouting System

Agar tidak mengulang-ulang script yang sudah ada dan agar tidak ribet saat ada perubahan.

- bikin folder di **resource > views > layouts > main.blade.php**
- **main.blade.php** sebagai **layout utama**, dimana pada layout utama ini yang berbeda hanya isi container-nya saja.
- ganti isi halaman (container), dengan `@yield('container')`
- sekarang kita harus kasih tahu **halaman child-nya**
- hapus semua code halaman child home (home.blade.php) kecuali isi dari halaman home tersebut, isi seperti ini

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

@section('container')
    <h1>Hello, world!</h1>
@endsection
```

- lakukan hal yg sama pada halaman yang lainnya seperti about dan post
- Jika ingin component tertentu di pisah seperti navbar, kita bisa bikin folder **views > partials**
- bikin file **navbar.blade.php** hanya untuk menyimpan navbarnya saja
- cut code yg berisi navbar saja yg ada pada main, dan pastekan di navbar.blade.php

- sisipkan @include('partials.navbar') pada halaman main.
- ada cara lain untuk menyisipkan component tersendiri jika kita melihat dokumentasi laravel-nya, namun kita menggunakan yg simple saja menggunakan directive-nya blade seperti @include, @yield, dll.

Title dynamis sesuai halaman yang aktif

- agar title website berubah sesuai dengan halaman yg aktif, masuk ke **resource > views > layouts > main.blade.php**

```
<title>Faiz Blog | Home</title>
```

hal ini menyebabkan semua title tetap bertuliskan "Home" meskipun kita berpindah-pindah halaman..

- agar title dapat berubah sesuai dengan halaman yg aktif, kita masuk ke **routes > web.php**
- kita kirimkan judul halaman ke masing-masing view nya menjadi seperti ini

```
Route::get('/', function () {
    return view('home', [
        "title" => "Home"
    ]);
});

Route::get('/about', function () {
    return view('about', [
        "title" => "About",
        "name" => "Akhmad Faiz Abdulloh",
        "email" => "akhmadfaizabdulloh@gmail.com",
        "image" => "cute.jpeg"
    ]);
});

Route::get('/blog', function () {
    return view('posts', [
        "title" => "Posts"
    ]);
});
```

- jika sudah, pada main.blade.php kita ganti menjadi

```
<title>Faiz Blog | {{ $title }}</title>
```

Navigasi Nyala saat halaman aktif

- Agar navigasi nyala saat halaman aktif, masuk ke **resource > views > partials > navbar.blade.php** ubah menjadi seperti

```
<ul class="navbar-nav">
    <li class="nav-item">
        <a class="nav-link" {{ ($title === "Home") ? 'active' : '' }}" href="/">Home</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" {{ ($title === "About") ? 'active' : '' }}" href="/about">About</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" {{ ($title === "Posts") ? 'active' : '' }}" href="/blog">Blog</a>
    </li>
</ul>
```

Simulasi blog posts (tanpa database)

- test
- untuk membuat tulisan lorem, buat di halaman view (posts.blade.php)
- membuat variable \$blog_posts pada route
- jika sudah ada datanya, datanya sudah bisa di akses di view posts (posts.blade.php)
- jika ingin mengecek isi datanya kita bisa dump menggunakan directive milik blade yang namanya @dd()

- @dd () => dump and die
- untuk melihat dari variable/object/array
- sama seperti var_dump tapi menggunakan style laravel

```
@dd($posts)

@extends('layouts.main')
@section('container')
    <h1>Halaman Posts</h1>
@endsection
```

Menggunakan Looping milik blade

- @foreach(...)

```
@foreach ($posts as $post)
    <article class="mb-5">
        <h2>{{ $post["title"] }}</h2>
        <h5>By: {{ $post["author"] }}</h5>
        <p>{{ $post["body"] }}</p>
    </article>
@endforeach
```

- disini kita bisa pakai foreach() untuk me-looping isi dari array, meskipun berupa array manual.

Single Post

- membuat sebuah view lagi, dan sebuah route lagi untuk mengakses salah satu post saja
- sekarang bungkus title dengan tag a

```
<h2>
    <a href="/posts/{{ $post["slug"] }}">{{ $post["title"] }}</a>
</h2>
```

- tambahkan slug pada route/web.php

```
"slug" => "judul-post-pertama",
```

- buat routes baru untuk halaman single post

```
// halaman single post
Route::get('posts/{slug}', function ($slug) {
    return view('post');
});
```

- membuat views baru "post.blade.php"

```
{-- @dd($post); --}
@extends('layouts.main')

@section('container')
<article>
    <h2>{{ $post["title"] }}</h2>
    <h5>{{ $post["author"] }}</h5>
    <p>{{ $post["body"] }}</p>
</article>

<a href="/blog">Back To Posts </a>
@endsection
```

- kirimkan data title dulu agar bisa di akses halamannya

```
return view('post', [
    "title" => "Single Post"
]);
```

- copy kan array \$blog_posts [] yang berisikan judul, author dan isi artikel, masukkan ke dalam routes halaman single post

```
$blog_posts = []
```

- untuk mencari postingan yg slug-nya sama dengan yg ada di halaman posts dengan menggunakan foreach()

```
$new_post = [];
foreach ($blog_posts as $post) {
    if ($post["slug"] === $slug) {
        $new_post = $post;
```

```
        }

    }

return view('post', [
    "title" => "Single Post",
    "post" => $new_post
]) ;
```

- Cara penggunaan blade templating engine (DONE)

5. Model, Collection & Controller

Di video sebelumnya,

di aplikasi sederhana kita itu semua proses-nya masih ditangani oleh komponen routes kita

contoh-nya ketika kita melakukan request ke halaman post untuk menampilkan semua data dari blog post kita, itu dilakukan didalam route.

begitu pula dengan proses menampilkan halaman view, itu juga dilakukan didalam route.

Nah, hal ini kurang tepat. Karena kalau misal kita mau **menerapkan konsep MVC**, 2 proses tersebut harus kita pisahkan sesuai komponen-nya.

Kalo misalkan kita mau **bekerja dengan data**, itu kita harus simpan codingan-nya **dalam MODEL**.

Dan kalau misalkan **ada proses** seperti memilih dan menampilkan view, itu biasanya dilakukan **di dalam controller**.

Kita akan coba mengimplementasikan / menerapkan konsep MVC dengan memisahkan code-nya kedalam komponen-komponen terpisah.

Jadi kita akan belajar membuat Model, membuat Controller, dan bagaimana cara menggabungkan ketiga komponen MVC-nya

Memperbaiki code video sebelumnya

Ada titik koma (;) yang muncul di tampilan, itu terjadi ketika kita panggil partial kita.

Harusnya syntax-nya blade itu tidak perlu diakhiri dengan titik koma (;)

Untuk me-navigasi antar file di dalam laravel menggunakan vscode, kalau mau cepet dan kita sudah tau nama file-nya apa. (bahkan kita tidak perlu buka buka lagi sidebar navigator).

Kita cukup pencet **Ctrl/Command + p > ketik nama file yang ingin dibuka** (semisal web.php)

Pada halaman blog./post sebenarnya tidak ada masalah.

Kita masih belum ada database-nya

dan kita masih kirim data-nya melalui view yang disimpan di route (web.php)

Kita akan perbaiki seolah-olah kita akan menggunakan sebuah model di dalam laravel.

Yang kita lakukan sebelumnya kurang tepat karena kita mengulang-ulang datanya. Kalau misalkan data yang pertama kita ubah isi nya, maka data yang kedua pun harus kita ubah juga isinya.

Apa itu Model?

Dokumentasi Model di MVC nya laravel :

<https://laravel.com/docs/8.x/eloquent#generating-model-classes>

Eloquent Model merupakan fitur yang ada di dalam laravel yang merupakan ORM (Object Relational Model)

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

Laravel menyertakan Eloquent, pemeta objek-relasional (ORM) yang memudahkan interaksi dengan basis data Anda. Saat menggunakan Eloquent, setiap tabel basis data memiliki "Model" terkait yang digunakan untuk berinteraksi dengan tabel tersebut. Selain mengambil data dari tabel basis data, model Eloquent juga memungkinkan Anda memasukkan, memperbarui, dan menghapus data dari tabel.

Cara membuat model

kita bisa menggunakan Artisan

```
$ php artisan make:model NamaModel
```

Nanti kalau udah ada tabel-nya, bahkan kita bahkan bisa bikin migration-nya

```
$ php artisan make:model NamaModel --migration
```

* Migration untuk menuliskan skema database, dan juga nanti akan meng-generate menjadi tabel didalam database-nya.

Implementasi ke aplikasi kita

Kita bisa membuat Model menggunakan Command Pallet milik VSCode jika kita sudah menginstall **Extension** yang namanya **Laravel Artisan (by Ryan Naddy)**

Untuk membuka command pallet

Ctrl/Command + Shift + P

kemudian ketikkan

```
>>> Artisan: Make Controller
```

Kemudian kita tulis nama controller-nya apa... misal : **Post**

(karena kita mau bikin model yang mengelola data postingan, dan nanti kita mau bikin tabel yang namanya Post juga.)

kemudian pilih **No** dulu semua-nya

otomatis dibuatkan file Post.php di dalam folder Model.

Atau kita pake plugin, bisa membuat model dengan perintah di **vscode**, dengan cara **ctrl + shift / command + P** untuk mengeluarkan command paletnya. kemudian ketikkan

```
>>> Artisan: Make Controller
```

Kemudian kita tulis nama controller-nya apa... misal : **LoginController** kemudian pilih tipe yang **Basic**

otomatis dibuatkan di dalam folder controller.

Sekarang kita simpan data yang ada di route (web.php) ke dalam Model-nya
app > Models > Post_.php

```
class Post
{
    private static $blog_posts = [
        [
            "title" => "Judul Post Pertama",
            "slug" => "judul-post-pertama",
            "author" => "Sandhika Galih",
            "body" => "Lorem ipsum 100"
        ],
        [
            "title" => "Judul Post Kedua",
            "slug" => "judul-post-kedua",
            "author" => "Doddy Ferdiansyah",
            "body" => "Lorem ipsum 100"
        ],
    ];
}
```

Install **Extension** yang namanya **PHP Namespace Resolver (by Mahedi Hassan)** untuk memudahkan mengimport namespace/class yang diperlukan

```
public static function all()
{
    return collect(self::$blog_posts);
}

public static function find($slug)
{
    $posts = self::$blog_posts;
    $post = [];
    foreach ($posts as $p) {
        if ($p["slug"] === $slug) {
            $post = $p;
        }
    }

    return $posts;
}
}
```

Dan sekarang, single post/ detail post-nya sudah ngambil dari Model.
Sekarang routes-nya lebih rapi, tidak ada data. Karena data harusnya memang di Model

Dan ini masih pura-pura representasi data blog-nya (yang nanti bisa kita ambil dari database atau API)

kita ambil ke **method all()** untuk ngedapetin semua datanya

```
public static function all()
{
    return collect(self::$blog_posts);
}
```

Kalau mau nyari satu postingan saja, kita ambil dulu semua data-nya di method `find()` kemudian kita looping.

```
public static function find($slug)
{
    $posts = self::$blog_posts;

    $post = [];
    foreach ($posts as $p) {
        if ($p["slug"] === $slug) {
            $post = $p;
        }
    }

    return $post;
}
```

Dan ini alasan kenapa kita harus menarik data ke dalam Mode kita.

Collection di Laravel

Selanjutnya, kita bisa mengubah data yang kita ambil menjadi sesuatu yang disebut dengan Collection di laravel.

Apa itu Collection?

Collection sebetulnya adalah pembungkus untuk sebuah array, yang akan membuat array menjadi **LEBIH SAKTI**.

Jadi kita kan punya `$blog_posts` yang isinya array di dalam array
Ada Array numerik yang di dalamnya ada array asosiatif.
Kita akan bungkus array biasa (asosiatif) menjadi sebuah colecion.

Dokumentasi Collections :

<https://laravel.com/docs/8.x/collections#introduction>

```
public static function find($slug)
{
    $posts = static::all();

    return $posts->firstWhere('slug', $slug);
}
```

Biasanya blog memiliki kategori masing-masing

Kita mulai dari dengan membuat model dan migrasinya untuk “Kategori”

```
$ php artisan make:model -m Category
```

* **flag “-m”** artinya bikin migration nya sekalian.

[Update terakhir, video ke - 5 menit ke 19:22]

6. Database, Migration & Eloquent

Environment Variable

Dokumentasi :

<https://laravel.com/docs/8.x/configuration#environment-configuration>

To make this a cinch, Laravel utilizes the [DotEnv](#) PHP library. In a fresh Laravel installation, the root directory of your application will contain a .env.example file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to .env.

- Didalam laravel ada yang namanya file **.env**
- File **.env** digunakan untuk melakukan setting pada lingkungan pengembangan aplikasi kita
 - karena nantinya kita tidak akan menyimpan/mendistribusikan konfigurasi ini
 - karena konfigurasi harusnya hanya diketahui oleh developernya aja, tidak bisa dilihat oleh orang lain , apalagi ketika aplikasi kita sudah di upload baik itu ke web hosting ataupun ke kode repository kita
- **.env.example** = template
- **.env** = variable sebenarnya yg digunakan oleh aplikasi kita
 - pada saat install laravelnya, sudah ada nilai default yg diisikan. contohnya seperti APP_KEY
 - APP_NAME=WPU_Blog
 - APP_ENV=local (untuk ngasih tau lingkungan pengembangan kita sekarang lagi proses tahap apa?). ada local, development, ada production
 - kalo misalkan kita gunakan production, nanti perilaku laravelnya akan berubah. misalnya tidak menampilkan pesan kesalahan selengkap klo kita ubah dia jadi local.
 - lalu juga ada perubahan lain yg intinya ngasih tau bahwa aplikasi kita siap untuk production, siap di akses atau dirilis
- konfigurasi database (default mysql)

- intinya file env ini agar laravelnya bisa terhubung ke lingkungan pengembangan kita sekarang. dan nanti kalo aplikasinya udah siap di publish atau di deploy atau di upload ke internet. file env ini tidak akan kita bawa. apalagi di simpan ke code repository. maka dari itu sudah disiapkan file .gitignore oleh laravelnya.
 - jadi nanti semisal sudah di deploy, kita akan buat lagi file .env di tempat penyimpanan website aslinya.
- jika kita masuk ke folder config/database.php akan ada konfigurasi database

```
'default' => env('DB_CONNECTION', 'mysql'),
```

konfigurasi defaultnya dia manggil fungsi yg namanya env() parameter pertamanya 'DB_CONNECTION' parameter keduanya 'mysql'

artinya, secara default koneksi database yg digunakan laravelnya apapun yg ada di dalam 'DB_CONNECTION' pada file .env. Jika tidak ada, atau tidak bisa koneksi ke .env, maka akan ada nilai default yg digunakan yaitu 'mysql'.

Persiapan MySQL

- pastikan koneksi servicenya database nya sudah jalan.
- bisa menggunakan phpmyadmin atau MySQL Workbench. atau Sequel Pro
- download di MySQL Workbench di websitenya
<https://www.mysql.com/products/workbench/>

Atau bisa menggunakan **phpmyadmin**

- sesuaikan sistem operasi dan versinya yg bener-bener support.
- Tambahkan MySQL Connection
 - Hostname : 127.0.0.1
 - Port : 3306
 - Username : root
 - Password : -

Tinggal kita update file .env

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

- Buat Database / Schema baru

Schema Name : wpu_blog (default)

Database : Migration

- Selalu baca dokumentasinya dulu
- Dokumentasi : <https://laravel.com/docs/8.x/migrations#main-content>

#Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The Laravel Schema [facade](#) provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

- Migration : seperti Version control untuk database kita. kayak git, bisa melacak perubahan yg ada pada codingan kita.
nah..Migration ini untuk melacak perubahan yang terjadi pada database kita lewat codingan laravel.

- Untuk melakukan migrate bisa menggunakan terminal di vscode atau terminal terpisah (pastikan masuk ke dalam folder aplikasinya)

```
$ php artisan migrate
```

- Jika berhasil, akan ada 3 tabel baru pada database beserta tabel migrations (anggap aja seperti folder .git). tabel inilah yang akan melacak perubahan pada database.
- Tabel-tabel ini muncul karena ada file didalam folder **database/migrations/**
- Jadi kalo kita mau bikin tabel baru, kita bikin file migration baru
- kita lihat file migrations yg users
- didalam file migration, pasti ada 2 buah method di dalam class-nya
- ada method yg namanya UP, ada method yang namanya DOWN

```
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {

    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

```
}
```

- method UP, adalah sebuah method yang akan kita gunakan ketika kita mau bikin schema/struktur dari tabelnya.

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Jadi kita ngga lagi pake DBMS nya untuk creatata table, strukturnya apa, panjang nya berapa, tipedatanya apa. semua kita tulis didalam kodingannya.

- method UP akan di eksekusi ketika kita ketik migrate

- method DOWN, kebalikannya.. untuk menghilangkan/menghapus schema yang udah kita bikin. perintahnya jika kita pakai php artisan, itu namanya rollback.

```
$ php artisan migrate:rollback
```

- Semua tabel hilang kecuali **tabel migrations**. karena inilah yg melacak perubahan tadi.

- kerenn banget buat kerja tim/mau deployment.. gaperlu import export data manual. tinggal kita lakukan semuanya di laravel

- Ada perintah yg bisa melakukan rollback dan migrate sekaligus.

```
$ php artisan migrate:fresh
```

- bisa digunakan saat kita mau mengubah skema dari tabel kita.
- laravel juga bakal tahu kalau semisal kita ngga sengaja melakukan migration di production. akan ada konfirmasi.
- semisal kita menambahkan satu buah field untuk mengecek apakah seorang user itu admin atau bukan.

```
$table->boolean('is_admin')->default(false);
```

Database : Eloquent ORM

- Ada cara dimana kita bisa ngisi data tapi ngga lewat mysql workbench/phpmyadmin, bahkan kita tidak perlu bikin dulu form di webnya.
- kita bisa melakukan itu dengan mudah, karna di laravel ada yg namanya eloquent.

Selalu baca dokumentasinya dulu

- Dokumentasi : <https://laravel.com/docs/8.x/eloquent#database-connections>

#Introduction

Laravel includes **Eloquent**, an **object-relational mapper (ORM)** that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

- Eloquent merupakan ORM (sebuah fungsi untuk memetakan tiap-tiap data yg ada di dalam tabel/database nanti kedalam sebuah objek) yang membuat kita dapat mudah berinteraksi dengan database yg kita buat.
- Setiap kita menggunakan Eloquent ini, setiap tabel di dalam database kita itu berkorespondensi atau terhubung kedalam sebuah "**Model**" yang bisa kita gunakan untuk berinteraksi dengan tabel tadi.
- Jadi antara website kita dan tabel didalam mysql kita ada perantara yg namanya "**Model**"

- Model yang sebelumnya kita buat belum sempurna karena masih belum terhubung ke tabel, kita hanya bikin pura-puranya saja.
- Selain kita bisa mengambil record/data2 dari tabel di database kita, Eloquent model juga memungkinkan kita untuk melakukan CRUD kedalam tabel lewat codingan. Jadi kita tidak perlu menyentuh database client lagi
- dan ini bisa terjadi karena yg namanya **Active Record Pattern**.

https://en.wikipedia.org/wiki/Active_record_pattern

The active record pattern is an approach to accessing data in a **database**. A **database table** or **view** is wrapped into a **class**. Thus, an **object** instance is tied to a single row in the table. After creation of an object, a new row is added to the table upon save. Any object loaded gets its information from the database. When an object is updated, the corresponding row in the table is also updated. The wrapper class implements **accessor methods** or properties for each column in the table or view.

This pattern is commonly used by object persistence tools and in **object-relational mapping** (ORM). Typically, **foreign key** relationships will be exposed as an object instance of the appropriate type via a property.

- Jadi *Active record* adalah salah satu pendekatan untuk membaca data dari sebuah **basis data**. Sebuah **tabel** atau **view** dibungkus ke dalam sebuah **kelas** sedemikian sehingga sebuah instansi **objek** terikat dengan satu baris tunggal dalam tabel. Setelah objek dibuat, sebuah baris baru ditambahkan pada tabel pada saat disimpan. Objek-objek yang dimuat mendapatkan informasi tentang dirinya dari basis data; ketika sebuah objek diubah, baris yang berkaitan dalam tabel juga diubah. **Kelas pembungkus** mengimplementasikan metode-metode aksesor untuk setiap kolom di dalam tabel atau **view**.

- Untuk melakukan itu, kita bisa menggunakan **Tinker**
- Tinker adalah sebuah aplikasi didalam laravel yang bisa kita gunakan untuk berinteraksi dengan aplikasi laravel kita
- caranya dengan mengetikkan perintah “`php artisan tinker`” pada terminal didalam folder aplikasi kita.
- Kalau kita buka folder **App/Models/** otomatis akan ada file **User.php** yang merupakan file model yang **pasangan** dari file **database/migrations/2014_10_12_000000_create_users_table**

- pada file **App\Models\User.php** akan ada

```
protected $fillable = [  
    'name', 'email', 'password',  
];
```

\$fillable yang artinya fill-fill mana aja yang boleh diisi, dan sisanya akan diisi oleh laravelnya secara otomatis.

ada hidden, dll.

- Jadi ini kita anggap sebagai **class blue print** nya, cetakan ketika nanti kita mau bikin user baru. Jadi model sebagai representasi tabel di codingan kita.

- kalau kita mau bikin user baru, kita bikin sebuah variabel namanya user (`$user`) yang berisi instans dari class user sebagai model kita.

```
>>> $user = new App\Models\User;
```

atau jika menyimpan modelnya sudah sesuai aturan bisa langsung

```
>>> $user = new User;
```

dengan begitu, laravel akan berasumsi bahwa menyimpannya di **App\Models\User**

- untuk mengisi caranya begini :

```
>>> $user->name = 'Sandhika Galih';
```

ingat, yang bisa diisi cuma 3 tadi, yaitu : nama, email, dan password saja

```
>>> $user->email = 'sandhika@gmail.com'
```

tanpa titik koma (;) di akhir juga boleh

```
>>> $user->password = bcrypt('12345')
```

kita bisa langsung panggil fungsi **bcrypt()**

- dengan menuliskan ini artinya kita sudah menuliskan 1 object yang namanya user berisikan data tersebut
- namun data belum tersimpan ke dalam database

- agar tersimpan ke dalam tabel kita harus melakukan **save()**

```
>>> $user->save()
```

- kalau kita ingin melihat isinya, kita bisa menggunakan method/fungsi **all()**
- hasil yang dikembalikan berbentuk **Collection**

- kerennya, kalau sudah bicara mengenai **Collection**, kita bisa lakukan banyak hal. contohnya kalau kita mau cari user yang pertama (seperti pada video sebelumnya) kita tinggal tulis

```
>>> $user->first()
```

user dengan id = 2

```
>>> $user->find(2)
```

- jika kita nyari yang datanya tidak ada, contoh id 2000, maka nanti responnya **null**
- namun ada satu method yang keren untuk langsung menangani ketika yg kita cari tidak ada, yaitu **findOrFail()**

```
>>> $user->findOrFail(2000)  
>>> $user->findorfail(2000)
```

```
Illuminate\Database\Eloquent\ModelNotFoundException  No query results  
for model [App\Models\User] 2000.
```

Jadi responnya sebuah **Exception**. Enak banget ketika kita mau pakai pesan kesalahannya.

7. Post Model

Membuat Model Post

Kita akan memperbaiki **Models Post** yang sudah kita buat secara manual, yang berisi method yang juga manual. padahal didalam **collection** laravel kita bisa gunakan all() dan find() bahkan lebih banyak lagi.

- kita hapus atau kita rename menjadi Post_.php model yang sebelumnya (agar tidak hilang sebagai history belajar kita)
- kita bikin lagi modelnya, tidak lagi bikin file manual, tapi bisa menggunakan perintah artisan dengan terminal.

```
$ php artisan make:model
```

setelah membuat model, kita bikin migrationnya. untuk membuat skema tabel post kita. jadi kita bikin 2 kali.

Tapi ada caranya agar bisa membuat model dan migrasinya sekaligus satu kali command.

- Jika masih bingung perintah apa yang bisa dilakukan oleh artisan, kita bisa menambahkan keyword help di depan.

```
$ php artisan help make:model
```

kita juga bisa membuat model dengan perintah di **vscode**, dengan cara **ctrl + shift / commad + P** untuk mengeluarkan command paletnya. kemudian ketikkan

```
>>> artisan: make model
```

nantinya sekaligus migrasinya otomatis dibuatkan.

Jika ingin membuatnya dengan terminal berikut caranya :

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

nanti akan dibuatkan 2 buah file. yang pertama modelnya, kedua migrasinya dengan nama tabel yang sudah plural (jamak) = Posts.
Sedangkan nama model kita itu singular = Post

```
Model created successfully.  
Created Migration: 2024_04_13_143059_create_posts_table
```

Jika kita kembali ke codingan kita, maka di dalam folder **model** kita sudah ada file **Post.php** dan udah include semua yang kita butuhkan.

app > Models > Post.php

dan juga yang penting pada folder **database > migrations** sekarang nambah 1 file lagi, yaitu **2024_04_13_143059_create_posts_table.php**

Kita buka file migrasinya (**2024_04_13_143059_create_posts_table.php**), untuk membuat skema databasenya di file tersebut.

kita cari method UP dimana kita membuat *blue print* nya disitu

field yang kita butuhin :

yang jelas butuh **id()** sebagai primary_key nya

kita juga butuh **timestamps()** untuk membuat field **created_at** dan **updated_at**

```
Schema::create('posts', function (Blueprint $table) {  
    $table->id();  
    $table->timestamps();  
});
```

kita bikin di tengah-tengah nya, seperti ini :

```
public function up()
```

```
{  
    Schema::create('posts', function (Blueprint $table) {  
        $table->id();  
        $table->string('title');  
        $table->text('excerpt');  
        $table->text('body');  
        $table->timestamp('published_at')->nullable();  
        $table->timestampls();  
    }) ;  
}
```

keterangan :

- tipe data “text” untuk data yang lebih dari 225 karakter
- **timestamp** (tanpa s) berbeda dengan dengan **timestampl** (method untuk membuat created_at updated_at)
- **nullable()** artinya boleh kosong.

lalu running :

```
$ php artisan migrate:fresh
```

isikan data tersebut dengan tinker

```
$php artisan tinker  
  
//bikin instance untuk setiap postnya  
  
>>> $post = new Post  
>>> $post->title = 'Judul Pertama'  
>>> $post->excerpt = ' lorem100 '  
  
//bikin dgn file baru (ctrl + N) dengan plaintext html "lorem100"  
  
>>> $post->body = ' lorem 100 full '  
  
>>> $post->save()
```

tambahkan 1 data lagi

```
>>> $post = new Post
```

```
>>> $post->title = 'Judul Kedua
>>> $post->excerpt = ' lorem80 '

// bikin dgn file baru (ctrl + N) dengan plaintext html "lorem100"

>>> $post->body = ' lorem 800 full '

>>> $post->save()
```

sekarang coba query isinya pake tinker

```
// panggil class modelnya -> Post::

>>> Post::all()
// untuk menampilkan semua data

>>> Post::first()
// untuk menampilkan data yg pertama

>>> Post::pluck('title')
// untuk menampilkan semua data, namun hanya title-nya saja

>>> Post::find(2)
// untuk mencari data yg id nya 2
```

Jika kita lihat di webnya, data nya akan otomatis tampil. Padahal belum kita apa-apain (view-nya, modelnya, dll)

pada model yang baru (**app > Models > Post.php**) tidak ada **method all()** maupun **find()**, tidak seperti pada model yang kita buat sendiri sebelumnya. Karna **method all()** dan **find()** sebetulnya **sudah ada di Laravel**.

Sehingga saat kita buka controller kita (**http > Controllers > PostController.php**), **method all() dan find() sudah bisa digunakan**.

pada **method find()** masih menggunakan **\$slug**, akan kita ganti menjadi id (karna **\$slug masih belum ada**).

```
// alternatif buka file cepat dengan Ctrl + P  
>>> tulis nama file yg ingin dibuka
```

buka file **posts.blade.php**

```
<a href="/posts/{{ $post["slug"] }}">{{ $post["title"] }}</a>  
  
// ganti "slug" dengan "id"
```

maka saat klik judul pada website akan menuju postingan yang dituju.

untuk kedepannya, kita tidak akan menggunakan notasi array. Tapi kita akan pakai notasi object. Dan collection bisa mengakses kedua notasi tersebut.

```
{{ $post["id"] }}  
  
// ganti notasi menjadi object  
  
{{ $post->id }}
```

Blade Escape Character

agar tag HTML dieksekusi, gunakan berikut

```
{!! $post->id !!}
```

Mass Assignment Exception

kita juga bisa menginputkan data dengan menggunakan method `create()`

```
Post::create([
    'title' => 'Judul Ke Empat',
    'excerpt' => 'lorem10',
    'body' => '<p>Lorem, ipsum </p><p>Cupiditate</p>'
])
```

namun, jika kita jalankan pada `tinker` akan muncul error

MassAssignmentException. karena laravel secara default tidak memperbolehkan banyak property dimasukkan ke dalam tabel.

```
Illuminate\Database\Eloquent\MassAssignmentException Add [title] to
fillable property to allow mass assignment on [App\Models\Post].
```

Kecuali, sudah kita atur di Post model kita.

```
class Post extends Model
{
    use HasFactory;

    protected $fillable = ['title', 'excerpt', 'body'];
}
```

Atau bisa menggunakan `guarded`

```
class Post extends Model
{
    use HasFactory;

    // apa saja yang boleh diisi
    // protected $fillable = ['title', 'excerpt', 'body'];

    // boleh di isi, kecuali ...
    protected $guarded = ['id'];
```

```
}
```

Setelah merubah Post model, silahkan **restart tinker** nya dengan **exit (Ctrl + D atau C)** kemudian **jalankan lagi**

Fillable atau guarded juga akan terpakai saat kita ingin mengupdate data atau lainnya.

```
> Post::find(3)->update(['title' => 'Judul Ketiga Berubah'])  
= true
```

atau bisa mengupdate data dengan where

```
> Post::where('title', 'Judul Ketiga Berubah')->update(['excerpt'  
=> 'excerpt post 3 berubah'])  
  
= 1
```

Route Model Binding

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

Saat menyuntikkan ID model ke rute atau tindakan pengontrol, Anda akan sering meminta database untuk mengambil model yang sesuai dengan ID tersebut. Pengikatan model rute Laravel menyediakan cara mudah untuk menyuntikkan instans model secara otomatis langsung ke rute Anda. Misalnya, alih-alih menyuntikkan ID pengguna, Anda dapat menyuntikkan seluruh User Instance model yang sesuai dengan ID yang diberikan.

Laravel secara otomatis menyelesaikan model Eloquent yang didefinisikan dalam rute atau tindakan pengontrol yang nama variabelnya yang bertipe sesuai dengan nama segmen rute. Misalnya:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

buka file **web.php**

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

file **PostController.php**

```
public function show(Post $post)
{
    return view('post', [
        "title" => "Single Post",
        "post" => $post
    ]);
}
```

variable yang ada di **web.php** dengan yg ada di **PostController.php** harus sama, yaitu **\$post**

Sekarang kita tambahkan **slug** kedalam skema database kita. buka file migrasi **2024_04_13_143059_create_posts_table**

```
Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->string('title');

    $table->string('slug')->unique();

    $table->text('excerpt');
    $table->text('body');
    $table->timestamp('published_at')->nullable();
    $table->timestamps();
});
```

Slug tidak boleh sama karena menjadi url, maka dari itu kita gunakan unique()

Setelah itu kita lakukan migrasi lagi

```
$ php artisan migrate:fresh
```

Tambahkan data lagi menggunakan **Post::create([])**, akan **lolos mass assignment** karena kita menggunakan **\$guarded**

Kita cek di web sudah aman!

Buka lagi halaman view post, **posts.blade.php**

```
<a href="/posts/{{ $post->id }}>{{ $post->title }}</a>
```

Ubah route pada **web.php** menjadi { post:slug }. karena jika { post } saja, defaultnya akan mencari data berdasarkan "id"

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

8. Post Category & Eloquent Relationship

Keterhubungan antar tabel

di Laravel : Eloquent Relationship (hubungan antar model)

model kategori akan dihubungkan dengan model Post yang sudah kita buat.

Membuat Category

Biasanya blog memiliki kategori masing-masing

Kita mulai dari dengan membuat model dan migrasinya untuk “Kategori”

```
$ php artisan make:model -m Category
```

* flag “-m” artinya bikin migration nya sekalian.

Model = Singularnya

Migrasi = Plural (Jama’)

* Tabel belum di bikin

akan ada file **app > Models >Category.php**
dan juga file **database > migrations >**
2024_10_05_132900_create_categories_table.php

Bikin skema untuk kategori

buka file **2024_10_05_132900_create_categories_table.php**

```
Schema::create('categories', function (Blueprint $table) {
    $table->id();
    $table->string('name')->unique();
    $table->string('slug')->unique();
    $table->timestamps();
});
```

* kita bikin unique semua, karena sebagai identifier

Untuk menghubungkan antara **tabel Category** dan **tabel Post**
Kita harus bikin Foreign Key di dalam **migrasinya Post**

tambahkan 1 field lagi pada skema **2024_04_13_143059_create_posts_table.php**

```
Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->foreignId('category_id');
    $table->string('title');
    ...
});
```

Setelah itu kita eksekusi migrasi-nya lagi

```
$ php artisan migrate:fresh
```

Tambahkan kategori dengan tinker

```
$ php artisan tinker

//bikin instance untuk setiap category-nya

>>> $category = new Category
>>> $category->name = 'Programming'
>>> $category->slug = 'programming'
>>> $category->save()
```

Atau tambahkan kategori dengan **method create()**

```
$ php artisan tinker

>>> Category::create([
```

```
        'name' => 'Web Design',
        'slug' => 'web-design'
    ])
```

Namun sebelum itu kita atur dulu di Category model kita, dan restart tinker.

```
class Category extends Model
{
    use HasFactory;

    protected $guarded = ['id'];
}
```

Cek data kategori dengan tinker all()

```
$ php artisan tinker

>>> $category::all()
```

Sekarang kita coba isi postingan, yg connect manual dengan category yg ada

```
Post::create([
    'title' => 'Judul Pertama',

    'category_id' => 1,

    'slug' => 'judul-pertama',
    'excerpt' => 'Lorem, ipsum pertama',
    'body' => '<p>Lorem, ipsum </p><p>Cupiditate</p>'

])
```

Kalau misalkan kita mencari postingan yang kategori-nya “Programming”, caranya

```
$ php artisan tinker

>>> Post::where('category_id', 1)->get()
```

Data category_id yang ditampilkan dalam bentuk id, Bagaimana agar nama kategorinya yg muncul. Kalau pakai SQL biasa kita harus pakai JOIN antar tabel.

Kalau di **laravel**, kita harus tentukan dulu **RELATIONSHIP** antar tabelnya.

Eloquent Relationship

Dokumentasi :

<https://laravel.com/docs/8.x/eloquent-relationships#main-content>

Database tables are often related to one another. For example, a blog post may have many comments or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports a variety of common relationships:

- [One To One](#)
- [One To Many](#)
- [Many To Many](#)
- [Has One Through](#)
- [Has Many Through](#)
- [One To One \(Polymorphic\)](#)
- [One To Many \(Polymorphic\)](#)
- [Many To Many \(Polymorphic\)](#)

Artinya :

Tabel basis data sering kali saling terkait. Misalnya, sebuah posting blog mungkin memiliki banyak komentar atau pesanan dapat dikaitkan dengan pengguna yang memesannya. Eloquent memudahkan pengelolaan dan penggunaan hubungan ini, dan mendukung berbagai hubungan umum:

Sebelumnya, kita harus paham mengenai **relasi antar tabel**, khususnya di MySQL. Seperti **One to One**, **One to Many**, **Many to Many**, dll.

Table Relationship

categories
id 🔑
name
slug

?

posts
id 🔑
title
slug
excerpt
body
published_at

bagaimana kita menghubungkan antara tabel “categories” dengan tabel “post”?

- yang pertama, kita harus hubungkan dari sisi **data definition**-nya dulu.
Dengan cara menambahkan sebuah **Foreign Key**.
- category_id (**Foreign Key**)
- Sehingga tabel “categories” bisa berelasi dengan tabel “post”
(pak sandhika bacanya selalu dari tabel yang menitipkan **Foreign Key**-nya = tabel “categories”)

Foreign Key

categories
id 🔑
name
slug

posts
id 🔑
category_id ↗
title
slug
excerpt
body
published_at

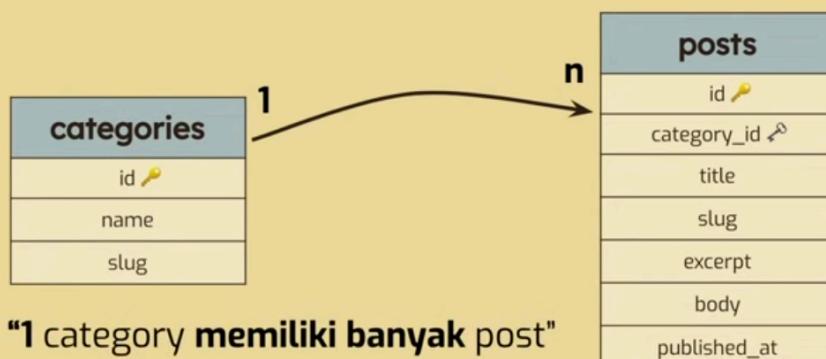
Berikutnya kita juga harus menentukan kardinalitasnya.

Cardinality

- One-to-One
- One-to-Many
- Many-to-Many

Berikutnya kita juga harus menentukan kardinalitasnya.

Cardinality



Kardinalitas juga punya kebalikan, di eloquent nyebutnya “inverse relationship”

Cardinality (inverse)

"1 post dimiliki oleh 1 category"

categories
id 🔑
name
slug

1 ← 1

posts
id 🔑
category_id ↗
title
slug
excerpt
body
published_at

Jadi, kalau dihubungkan dengan eloquent kita bacanya begini ...

Relationship

categories
id 🔑
name
slug

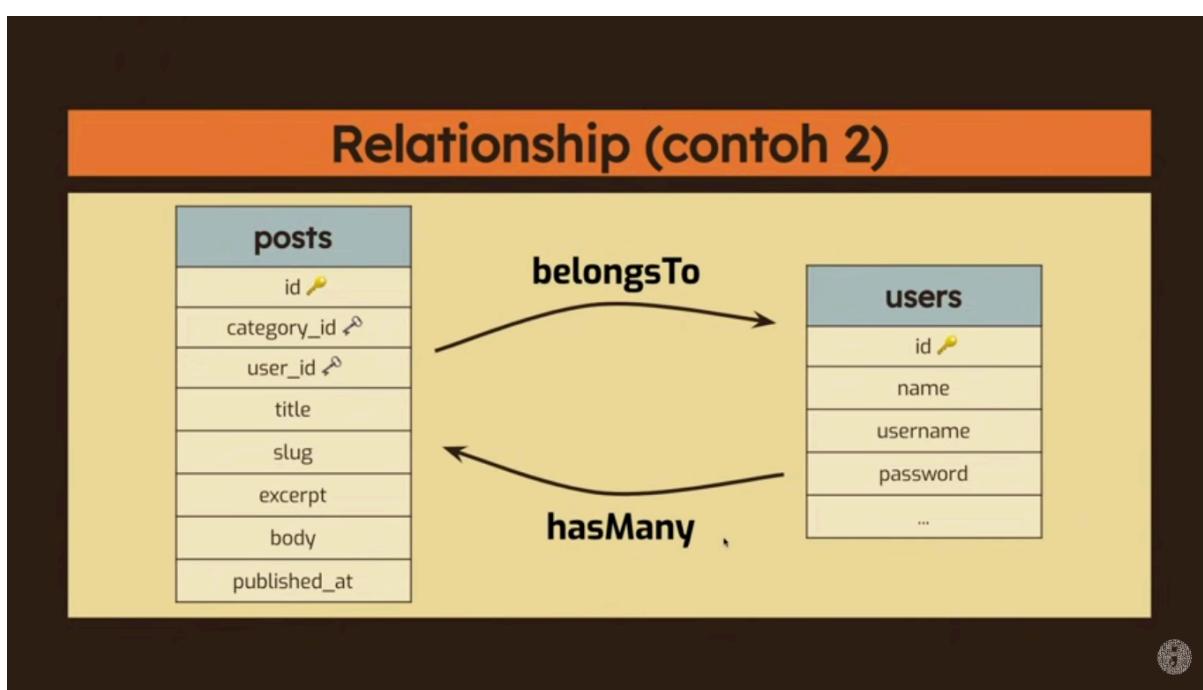
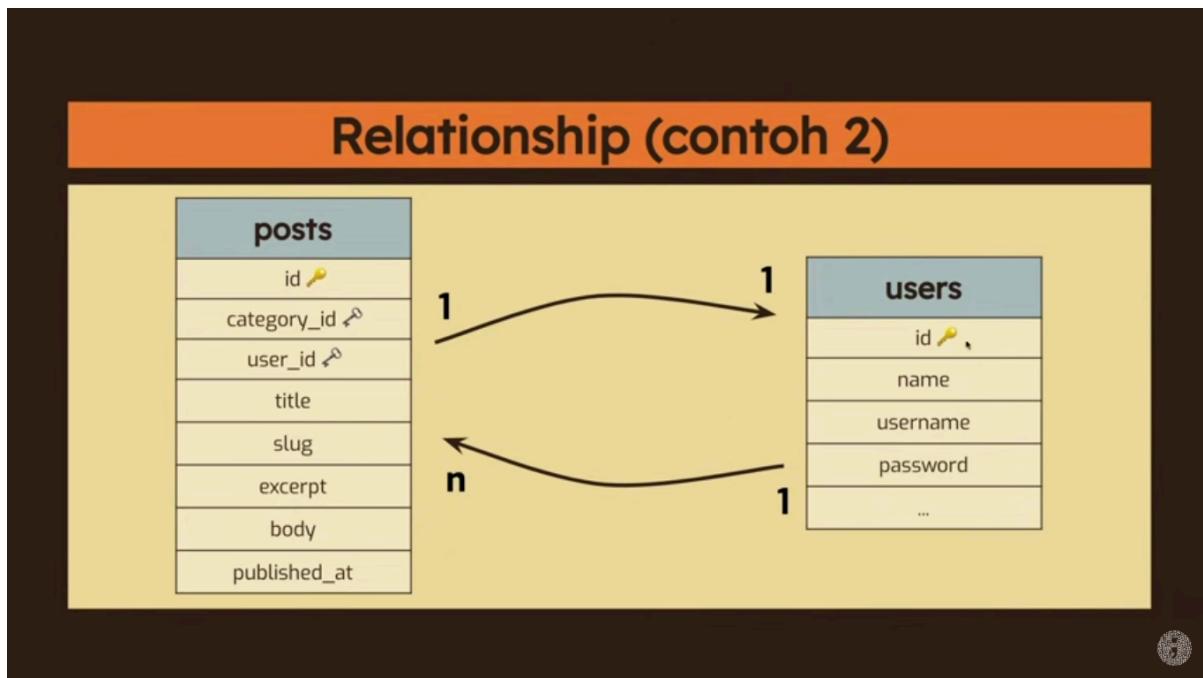
hasMany

← belongsTo

posts
id 🔑
category_id ↗
title
slug
excerpt
body
published_at

*Jika 1 ke 1, maka **hasOne**. Jika 1 ke banyak, maka **belongsToMany**

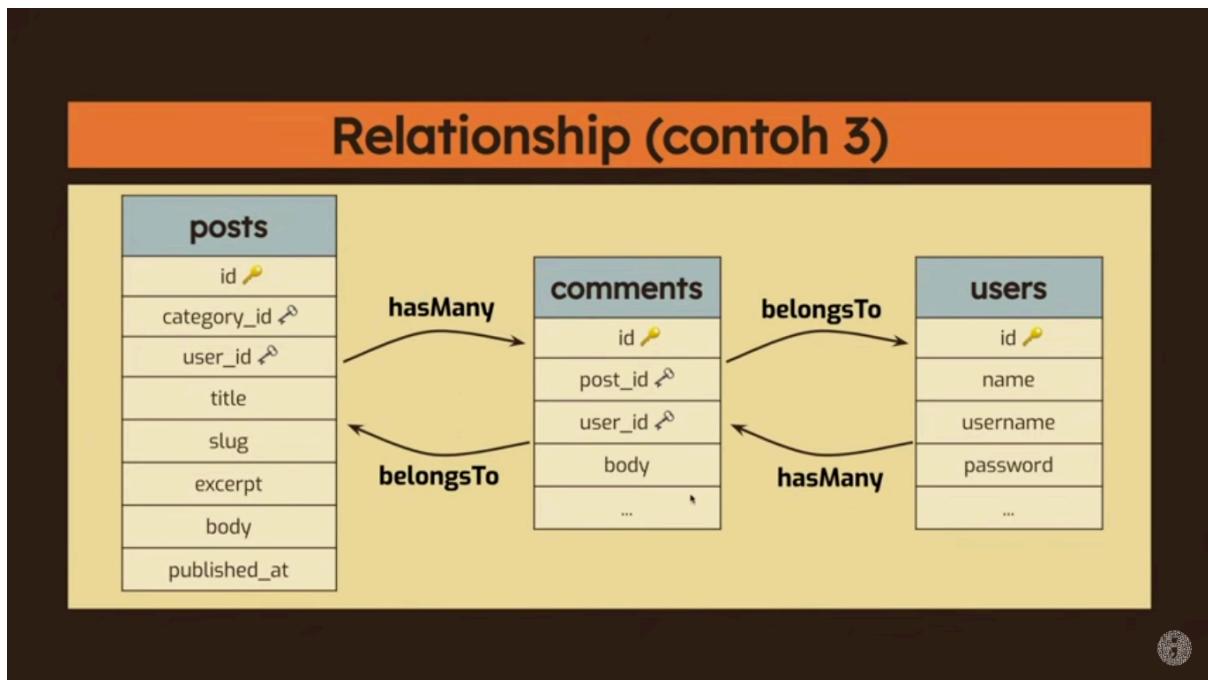
Contoh 2



*Cara penulisan Foreign Key defaultnya **versi singular** (tunggalnya) **dari tabel**. seperti **category_id / user_id**

*bisa pakai nama lain, namun kita harus ngasih tau ke laravel nya. Biar gampang langsung aja pakai **default**

Contoh 3



Implementasi Relationship



Dokumentasi :

<https://laravel.com/docs/8.x/eloquent-relationships#main-content>

Setelah kita tahu, relasi tabel kita apa.. OneToOne atau OneToMany.. kita bisa lihat contohnya di dokumentasi

Untuk menghubungkan Model Post dengan Model Category

kita buka lagi file **app > Models >Post.php**

di dalam class-nya kita harus **bikin** sebuah **method baru** dengan nama method yang sama dengan nama model yang akan dihubungkan.

Disini kita akan menghubungkan dengan Model Category

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

// belongsTo = 1 post hanya memiliki 1 kategori
```

*Dengan begitu, Model Post sudah berelasi dengan Model Category

Untuk mengecek apakah sudah benar-benar berelasi, kita bisa mengecek dengan tinker

```
$ php artisan tinker

>>> $post = $Post::first()
// melihat data postingan pertama

>>> $post->category
// menampilkan category dari postingan pertama di atas

>>> $post->category->name
// menampilkan nama category
```

Kita tambahkan nama penulis postingan dan kategori postingan
kita buka file View Single Post (tanpa s) **resources > views >post.blade.php**

```
<h1 class="mb-5">{{ $post->title }}</h1>

<p>By. Sandhika Galih in {{ $post->category->name }}</p>

{!! $post->body !!}
```

Kita buat agar kategorinya bisa diklik dan menampilkan semua postingan yang kategorinya tersebut.

```
<p>By. Sandhika Galih in <a href="/categories/{{ $post->category->slug }}">{{ $post->category->name }}</a></p>
```

Kita buatkan routes nya
buka file routes > web.php

```
Route::get('categories/{category:slug}', function (Category $category) {
    return view('category', [
        'title' => $category->name,
        'posts' => $category->posts,
        'category' => $category->name
    ])
});
```

Tambahkan Model Category agar tersambung dan tidak error (bisa juga ditambahkan otomatis oleh laravel jadi tidak perlu tambahkan sendiri)

```
use App\Models\Category;
```

Save As atau Copy resources > views >post.blade.php menjadi category.blade.php

ubah judul halamannya

```
<h1 class="mb-5">Post Category : {{ $category }}</h1>
```

Buka file Model Category app > Models >Category.php
kembalikan relasi untuk category ke post.
dengan menambahkan method seperti berikut

```
public function posts()
{
    return $this->hasMany(Post::class);
}
// hasMany = 1 kategori bisa di miliki oleh banyak post
```

Testing relasi dengan tinker

```
$ php artisan tinker

>>> $category = Category::first()
// mengambil kategori pertama = Programming

>>> $category->post
// menampilkan semua postingan yang kategorinya Programming
```

Membuat Halaman Categories

Kita bikin 1 route lagi agar ketika di akses <http://127.0.0.1:8000/categories/> agar di tampilkan semua kategori yang ada

buka file routes > web.php

```
Route::get('/categories', function() {
    return view('categories', [
        'title' => 'Post Categories',
        'categories' => Category::all()
    ]);
});
```

Save As atau Copy resources > views >category.blade.php menjadi categories.blade.php

9. Database Seeder

Biasanya kita menambahkan data dengan tinker

Saat dalam tahap pengembangan/development, kita masih sering mengubah-ubah skema dari tabel kita. Seperti menambah field baru, menghapus field yg sudah ada, nambah relasi, dll.

Setiap kita mengubah skema tabel, berarti kita harus melakukan kembali migrasi.

```
$php artisan migrate:fresh
```

yang artinya database kita akan kosong lagi, mulai lagi dari awal.

Kalau kita mau nambahin data lagi, kita buka tinker lagi, kita isikan lagi datanya manual, satu persatu.

Disinilah mulai terasa sangat ribet/sangat merepotkan.

Dengan begitu kita akan bahas Seeder dan Factory yang nantinya akan mempermudah kita untuk **mempopulasi** /mengisikan data secara otomatis saat kita melakukan migrasi.

Menambahkan link author & Menambahkan relasi di tabel post

Update tampilan **resources > views >posts.blade.php**

```
<article class="mb-5 border-bottom pb-4">
    <h2>
        <a href="/posts/{{ $post->slug }}"
class="text-decoration-none">{{ $post->title }}</a>
    </h2>

    <p>By. Sandhika Galih in <a href="/categories/{{ $post->category->slug }}"
class="text-decoration-none">{{ $post->category->name }}</a></p>

    <p>{{ $post->excerpt }}</p>
```

```
        <a href="/posts/{{ $post->slug }}"  
class="text-decoration-none">Read more..</a>  
  
    </article>
```

Untuk **menghubungkan** antara **tabel User** dan **tabel Post**
Kita harus bikin Foreign Key di dalam **migrasinya Post**

tambahkan 1 field lagi pada skema **2024_04_13_143059_create_posts_table.php**

```
Schema::create('posts', function (Blueprint $table) {  
    $table->id();  
    $table->foreignId('category_id');  
    $table->string('title');  
    ...
```

*duplikat baris di vscode dengan Alt + Shift + Bawah

Setelah itu kita eksekusi migrasi-nya lagi

```
$ php artisan migrate:fresh
```

Apa itu database seeding?

Dokumentasi :

<https://laravel.com/docs/8.x/seeding#main-content>

Laravel includes the ability to seed your database with data using seed classes. All seed classes are stored in the `database/seeds` directory. By default, a `DatabaseSeeder` class is defined for you. From this class, you may use the `call` method to run other seed classes, allowing you to control the seeding order.

Laravel menyediakan kemampuan untuk menyemai data ke dalam database Anda menggunakan kelas-kelas seed. Semua kelas seed disimpan dalam direktori database/seeder. Secara default, kelas DatabaseSeeder ditetapkan untuk Anda. Dari kelas ini, Anda dapat menggunakan metode call untuk menjalankan kelas-kelas seed lainnya, yang memungkinkan Anda untuk mengontrol urutan penyemaian.

seperti menyemai benih
seed = biji /benih
dengan seed ini, kita bisa mengisi otomatis tabel kita ketika kita buat (panen)

Jadi pahami dulu bahwa, Seeder merupakan fitur yg dikasih oleh laravel untuk meng-generate/mempopulasi isi dari tabel kita secara otoamatis dengan data testing/data dummy pada saat development.

untuk membuatnya kita bisa jalankan

```
$ php artisan make:seeder UserSeeder
```

Ini yang sudah dikasih ke kita.

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

```
    }  
}
```

Using Model Factories

Atau kita bisa pakai Factories (Kita bikin **pabrik pembuat data** untuk melakukan seeding nya secara otomatis)

untuk membuatnya kita bisa jalankan

```
php artisan db:seed  
  
php artisan db:seed --class=UserSeeder  
  
php artisan migrate:fresh --seed
```

Membuat Seeder

Buka file default seeder **database > seeders >DatabaseSeeder.php**

Kita coba aktifkan baris yang di komentar

```
public function run()
{
    // User::factory(10)->create();
    \App\Models\User::factory(10)->create();
}
```

kita coba jalankan

```
$ php artisan db:seed
```

Maka akan ada 10 data random di tabel users.

Jadi kita barusan melakukan seeding dengan bantuan factory.

Kita ilangin lagi datanya

```
$ php artisan migrate:fresh
```

Membuat Seeder Manual

Buka file default seeder **database > seeders >DatabaseSeeder.php**

Apa yang kita lakukan di tinker sebelumnya, **>>> Post::create()** akan kita lakukan di dalam **seeder**.

Kita tambahkan dulu

```
use App\Models\User;
use App\Models\Category;
use App\Models\Post;
```

Kemudian kita buat isiannya

```
User::create([
    'name' => 'Akhmad Faiz Abdulloh',
    'email' => 'akhmadfaiz@gmail.com',
    'password' => bcrypt('12345')
]) ;

Category::create([
    'name' => 'Web Programming',
    'slug' => 'web-programming'
]) ;

Category::create([
    'name' => 'Personal',
    'slug' => 'personal'
]) ;

Post::create([
    'title' => 'Judul Pertama',
    'category_id' => 1,
    'user_id' => 1,
    'slug' => 'judul-pertama',
    'excerpt' => 'Lorem, ipsum pertama',
    'body' => '<p>Lorem, ipsum100</p>'
]) ;
```

kita jalankan

```
$ php artisan db:seed
```

Catatan

Jika kita jalankan lagi **seeder-nya**, maka akan menambahkan kembali data yg ada di seeder.

```
$ php artisan db:seed
```

*Akan terjadi error karena ada **duplicat entry**, dan ada tabel yang harus tidak boleh ada yg sama (**unique**) seperti email dan slug.

jika ingin menambahkan data lagi, kita lakukan migrasi fresh, namun langsung kita kasih **flag-nya** “`--seed`”.

```
$ php artisan migrate:fresh --seed
```

Membuat Halaman Authors

Update tampilan resources > views >posts.blade.php

```
<p>By. <a href="#" class="text-decoration-none">{{  
$post->user->name }}</a> in <a href="/categories/{{  
$post->category->slug }}" class="text-decoration-none">{{  
$post->category->name }}</a></p>
```

Kita hubungkan Model Post dengan Model User
kita buka file Model Post-nya app > Models >Post.php

di dalam class-nya kita harus **bikin** sebuah **method baru** dengan nama method yang sama dengan nama model yang akan dihubungkan.

Disini kita akan menghubungkan dengan Model Category

```
public function user()  
{  
    return $this->belongsTo(User::class);  
}  
// 1 post hanya di miliki 1 user  
post hanya memiliki 1 kategori
```

kita buka file Model User-nya app > Models >User.php

```
public function posts()  
{  
    return $this->hasMany(Post::class);  
}  
// 1 user boleh punya banyak post
```

Kemudian kita testing dengan menambahkan 1 user baru lagi menggunakan seeder.

Kita perbaiki juga single post-nya **resources > views >post.blade.php**

```
<p>By. <a href="#" class="text-decoration-none">{{  
$post->user->name }}</a> in <a href="/categories/{{  
$post->category->slug }}" class="text-decoration-none">{{  
$post->category->name }}</a></p>  
  
{!! $post->body !!}  
  
<a href="/posts" class="d-block mt-3">Back To Posts </a>
```

10. Factory & Faker

Fitur Factory nanti akan kita gunakan ketika kita ingin membuat banyak data sekaligus secara otomatis untuk nanti kita jadikan sebagai data di dalam seeder kita.

Kalau sebelumnya kita menuliskan datanya secara manual satu persatu di dalam file seedernya.

Nanti kita juga akan menggunakan sebuah library yang dinamakan dengan **Faker** untuk membuat data kita otomatis terisi dengan data palsu/data dummy yang masuk akal (secara random).

Sehingga nanti kita tidak perlu memikirkan data apa yang akan kita masukkan sebagai seedernya.

Sebelumnya kita tulis secara manual dan sembarang, dengan **Faker** kita bisa meng-generate data palsu secara otomatis.

Apa itu Factory?

Factory = Pabrik (Pabrik pembuat data palsu)

Kalau kita masuk ke setiap **Model** (yang kita generate otomatis pake **artisan**) itu sudah sudah tersedia

```
use HasFactory;
```

jadi sebetulnya secara default, kita udah bisa bikin factory-nya

* Jika kita Ctrl + klik , kita akan masuk sebuah trait yang namanya factory.

Dokumentasi :

<https://laravel.com/docs/8.x/database-testing#defining-model-factories>

First, let's talk about Eloquent model factories. When testing, you may need to insert a few records into your database before executing your test. Instead of manually specifying the value of each column when you create this test data, Laravel allows you to define a set of default attributes for each of your [Eloquent models](#) using model factories.

To see an example of how to write a factory, take a look at the `database/factories/UserFactory.php` file in your application. This factory is included with all new Laravel applications and contains the following factory definition:

Pertama, mari kita bahas tentang pabrik model Eloquent. Saat melakukan pengujian, Anda mungkin perlu memasukkan beberapa catatan ke dalam basis data sebelum menjalankan pengujian. Alih-alih menentukan nilai setiap kolom secara manual saat Anda membuat data pengujian ini, Laravel memungkinkan Anda untuk menentukan serangkaian atribut default untuk setiap model Eloquent menggunakan pabrik model.

Untuk melihat contoh cara menulis pabrik, lihat file `database/factories/UserFactory.php` di aplikasi Anda. Pabrik ini disertakan dengan semua aplikasi Laravel baru dan berisi definisi pabrik berikut:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'name' => $this->faker->name(),
            'email' => $this->faker->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' =>
'$2y$10$92IXUNpkjO0rQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }
}
```

* isinya adalah sebuah class yang terhubung ke class factory

```
class UserFactory extends Factory
{
}
```

* dimana di dalamnya ada sebuah **method definition()** untuk bikin data user secara random

```
public function definition()
{
    return [
        'name' => $this->faker->name(),
        'email' => $this->faker->unique()->safeEmail(),
        'email_verified_at' => now(),
        'password' =>
'$2y$10$92IXUNpkjO0rQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
        'remember_token' => Str::random(10),
    ];
}
```

* Jadi kita bikin skema yang sama dengan skema yg kita bikin di Model.

* Hanya kita tentukan aja data-data mana yang boleh diisi.

* nama, dia manggil sebuah method yg namanya **faker->name()**

artinya : Generate nama palsu pakai library faker

* Generate email palsu yang udah safe (formatnya udah bener), yang dia unique pakai library faker

```
return [
    'name' => $this->faker->name(),
    'email' => $this->faker->unique()->safeEmail(),
];
}
```

Jadi di dalam Laravel sudah terintegrasi library faker.

Kita juga bisa bikin definisi kita sendiri untuk Model milik kita, atau bahkan kalau mau ngubah pun boleh.

Kita bisa bikin factory kita sendiri dengan

```
$ php artisan make:factory PostFactory
```

Apa itu Faker?

Dokumentasi :

<https://github.com/FakerPHP/Faker>

fakerphp.github.io atau <https://fakerphp.org/>

Sebenarnya kita bisa install manual kalau kita punya aplikasi yang bukan laravel dengan composer.

Namun di laravel sudah ada.

Installation#

Faker requires PHP >= 7.4.

```
$ composer require fakerphp/faker
```

Available Formatters#

<https://fakerphp.org/formatters/>

Jadi kita bisa meng-generate banyak data random, mulai dari gelar, nama, alamat, no. telepon, perusahaan.

Dan bisa sesuai apa yang kita butuhkan, bisa generate paragraf, kata-kata, dll.

Dan yang paling disukai ada adalah, dia punya locales-nya (localisation).

Dimana kita bisa bikin data-data yang kita punya, itu seperti data-data di negara kita (Indonesia).

Mengubah Faker menjadi Bahasa Indonesia

Buka file config > app.php

```
// 'faker_locale' => 'en_US',
'faker_locale' => env('FAKER_LOCALE', 'en_US'),
```

- * Default-nya Inggris (Amerika)
- * Kita bikin dia membaca file .env kita
- * Dia akan mengambil dulu yang ada di variabel **environment**, jika tidak ada maka defaulnya inggris

Buka file > .env

Tambahkan di paling bawah sendiri

```
FAKER_LOCALE=id_ID
```

- * Save dan kita ulangi seeding kita

Membuat data user menggunakan Factory

Buka lagi file seeder database > seeders > DatabaseSeeder.php

Sekarang, User tidak akan kita creat manual. kita Komentari (non-aktifkan)
User::create()

```
// User::create([
    //     'name' => 'Akhmad Faiz Abdulloh',
    //     'email' => 'akhmadfaiz@gmail.com',
    //     'password' => bcrypt('12345')
    // ]);
```

Kita akan bikin pakai **factory**, kita coba bikin 5 user

```
use App\Models\User; //pastikan namespace sudah di aktifkan  
  
User::factory(5)->create();
```

kita jalankan

```
$ php artisan migrate:fresh --seed
```

* Kita cek database kita sudah terisi 5 user dengan nama random orang indonesia.

Dan itulah Faker, dimana kita bisa meng-generate user secara random.
Ajaib yaa!

Membuat Factory untuk Post

Sekarang, kita akan bikin factory untuk kita sendiri, terutama untuk Post

Kita bisa pakai **command pallet** di vscode, dengan cara **ctrl + shift / command + P** kemudian ketikkan

```
>>> artisan: make factory
```

Atau bisa pakai terminal

```
$ php artisan make:factory PostFactory
```

* PostFactory = nama factory-nya
* diberi nama PostFactory agar sesuai dengan nama modelnya

Catatan tambahan!!!

Bahkan kedepannya kalau kita udah paham dengan factory, migrasi, seeder.
Pada saat kita bikin model bisa seperti ini
(semisal mau bikin data mahasiswa/Student)

```
$ php artisan make:model Student -mfs

keterangan flag
-m = untuk membuat migrasi-nya
-f = untuk membuat factory-nya
-s = untuk membuat seeder-nya
```

* Kita bisa kasih tambahan flag-nya -mfs
nantinya akan dibuatkan otomatis file migrasi, factory, dan seeder-nya
dan namanya sudah benar: creat_student, StudentFactory, dan StudentSeeder

Lanjut!

Setelah itu, kita cek harusnya sudah terdapat file factory post **database > factories >PostFactory.php**

kita isi seperti ini di **method definition()**

```
public function definition()
{
    return [
        'title' => $this->faker->sentence(mt_rand(2,8)),
        'slug' => $this->faker->sug(),
        'excerpt' => $this->faker->paragraph(),
        'body' => $this->faker->paragraph(mt_rand(5,10)),
        'user_id' => 1,
        'category_id' => 1
    ];
}
```

kita Komentari (non-aktifkan) atau hapus **Post::create()** yang ada di **database > seeders >DatabaseSeeder.php**

```
// Post::create([
    ...
    // ]);
});
```

kita buat **database > seeders >DatabaseSeeder.php** menjadi seperti ini

```
User::factory(5)->create();

Category::create([
    'name' => 'Web Programming',
    'slug' => 'web-programming'
]);

Category::create([
    'name' => 'Personal',
    'slug' => 'personal'
];

Post::factory(20)->create();
```

kita jalankan

```
$ php artisan migrate:fresh --seed
```

* Kita cek database atau web kita sudah terdapat data 5 user, 2 kategori, dan 20 post.

Kita update lagi factory untuk post **database > factories >PostFactory.php** menjadi seperti ini

```
public function definition()
{
    return [
        'title' => $this->faker->sentence(mt_rand(2,8)),
        'slug' => $this->faker->sug(),
        'excerpt' => $this->faker->paragraph(),
        'body' => $this->faker->paragraph(mt_rand(5,10)),
        'user_id' => 1,
        'category_id' => 1
    ];
}
```

kita migrate fresh lagi

```
$ php artisan migrate:fresh --seed
```

Amaann!

Jadi kita tidak perlu nulisin 20 postingan manual di tinker, **capek jugaa!**

Menampilkan data Post dari yang paling baru

Sekarang kita bisa bikin supaya data yang tampil itu yang terbaru duluan. Yang kita punya saat ini data terurut berdasarkan id.

kita buka Post controller kita **http > Controllers > PostController.php**
kita ubah menjadi seperti ini

```
public function index()
{
    return view('posts', [
        "title" => "Posts",
        // "posts" => Post::all()
        "posts" => Post::latest()->get()
    ]);
}
```

* dengan latest(), data yang terakhir di tambahin akan di tampilkan pertama/diatas

Namun, jika kita cek di web kita tidak akan ada perubahan. Karena semua postingan di buat di waktu yang sama, kita cek **created_at** semua sama persis di detik yang sama.

Kita coba bikin postingan baru

kita buka **database > seeders > DatabaseSeeder.php**
dan hanya aktifkan script ini saja, yang lainnya non-aktifkan

```
Post::factory(5)->create();
```

Namun, kemudian jangan di migrate fresh, namun cukup jalanin ini aja

```
$ php artisan db:seed
```

* Kita cek database dan web kita sudah terdapat tambahan 5 post dan sudah urut dari yang paling terbaru.

Menampilkan Post berdasarkan Author

Bagaimana menampilkan data postingan dari author itu saja.
Cara buatnya sebenarnya sama dengan yang kategori.

kita buka **routes** kita
routes > web.php

kita bikin route baru
bahkan kita ngga perlu view baru, kita bisa pinjem punya-nya Post.

```
Route::get('/authors/{user}', function (User $user) {
    return view('posts', [
        'title' => 'User Posts',
        'posts' => $user->posts,
    ]);
});
```

Update tampilan **resources > views > posts.blade.php**

```
<p>By. <a href="/authors/{{ $post->user->id }}"
class="text-decoration-none">{{ $post->user->name }}</a>
```

* Kita cek web kita sudah bisa melihat post dari **author** yang dipilih saja.
Namun masih menggunakan id
<http://127.0.0.1:8000/authors/1>

dan kita masih bisa melakukan tebak-tebakan id, dan itu kurang bagus.

Kita lihat di migrasi user kita
database/migrations/2014_10_12_000000_create_users_table

kita tidak bisa pakai name, karena tidak unique. Sedangkan untuk route model binding itu sebaiknya unik.

Kita bisa pakai email, tapi itu terlihat aneh.

Jadi kita bikin struktur baru, tambahin 1 field baru yaitu username

```
$table->string('username')->unique();
```

Jangan lupa!

Sebelum kita migrasi, kita perbaiki dulu factory-nya > **Userfactory.php**
username masih belum ada

```
'username' => $this->faker->unique()->userName() ,
```

* Faker juga punya **userName()**

Factory-nya beress!!

Kita bikin seedernya lagi. Sama seperti sebelumnya

```
User::factory(5)->create();

Category::create([
    'name' => 'Web Programming',
    'slug' => 'web-programming'
]) ;

Category::create([
    'name' => 'Personal',
    'slug' => 'personal'
]) ;

Post::factory(20)->create();
```

kita migrate fresh lagi

```
$ php artisan migrate:fresh --seed
```

kita benerin lagi **routes** kita
routes > web.php

```
Route::get('/authors/{user:username}', function (User $user) {
    return view('posts', [
        'title' => 'User Posts',
        'posts' => $user->posts,
    ]);
});
```

Update lagi Post view-nya **resources > views >posts.blade.php**

```
<p>By. <a href="/authors/{{ $post->user->username }}"
class="text-decoration-none">{{ $post->user->name }}</a>
```

* Kita cek web kita sudah bisa melihat post dari **author** dengan username
<http://127.0.0.1:8000/authors/kenari.nugroho>

Sekarang kita perbaiki lagi!

Karna konteknya author, bukan user jadi kita ubah route-nya

```
Route::get('/authors/{author:username}', function (User $author)
{
    return view('posts', [
        'title' => 'Author Posts',
        'posts' => $author->posts,
    ]);
});
```

Kita ubah semua yang user menjadi author, agar sesuai dengan studi case kita!

Kita buka lagi Model post kita **app > Models > Post.php**

```
public function author()
{
```

```
        return $this->belongsTo(User::class, 'user_id');
```

```
}
```

* Kalau kita langsung ubah method user() menjadi author, akan terjadi error.

Karna Laravel akan mengecek ada ngga di tabel Post, field yang namanya **author_id** sebagai **foreign** key-nya.

Padahal ngga ada, yang ada hanya **user_id**.

Untuk mengganti menjadi author(), kita tambahkan parameter 'user_id'

'user_id' aliasnya /nama lainnya author()

Terakhir kita Update lagi Post view dan Posts view-nya, user menjadi author.

resources > views >posts.blade.php

resources > views >post.blade.php

11. N+1 Problem

Jadi di video sebelumnya, kita sudah berhasil untuk melakukan relasi antar 3 tabel. Tabel Post, tabel kategori, dan juga tabel user. Untuk menampilkan Post berdasarkan kategori dan Post berdasarkan penulisnya/authornya.

Yang sudah kita lakukan ini itu kelihatannya tidak ada masalah, bahkan **SUDAH KEREN BANGET !!!** kita dapat dengan mudah menampilkan data di halaman web kita.

Tapi kita sebenarnya kita **menggunakan kesalahan yang cukup fatal !!!** dan akan berdampak pada performance aplikasi kita kedepannya.

Mungkin sekarang belum terasa, karena post yang kita punya masih belum banyak. Sekitar 20-25 post saja.

Tapi **bagaimana nanti kalo kedepannya semakin besar ?** post-nya udah banyak, penulisnya udah banyak, bisa sampai 200 post, bahkan 2000 post !!!

Nah, disitulah akan terasa performance dari aplikasi kita pasti menurun.

Problem yang kita hadapi itu dinamakan dengan **N+1 Problem**

Dan ini erat kaitannya dengan relasi serta query yang kita lakukan di aplikasi kita.

Nah, apa itu **N+1 Problem** dan gimana **cara mengatasinya?**

Memperbaiki halaman category

Setelah selesai kita membuat halaman author, kita akan memperbaiki dulu halaman kategori karena masih belum lengkap isi kontennya.

Sekalian kita bikin agar halaman kategori, author, dan posts akan mengarah ke view yang sama.

Jadi kita tidak perlu bikin view masing-masing lagi.

Jadi view Posts **resources > views >posts.blade.php** akan di pakai oleh ketiga-nya

kita buka lagi **routes** kita
routes > web.php

```
Route::get('/categories/{category:slug}', function (Category $category) {
    return view('posts', [
        'title' => "Post By Category : $category->name",
        'posts' => $category->posts
    ]);
});

Route::get('/authors/{author:username}', function (User $author) {
    return view('posts', [
        'title' => "Post By Author : $author->name",
        'posts' => $author->posts,
    ]);
});
```

Kita update juga post controller kita **http > Controllers > PostController.php**)

```
public function index()
{
    return view('posts', [
        "title" => "All Posts",
        // "posts" => Post::all()
        "posts" => Post::latest()->get()
    ]);
}
```

```
}
```

Sekarang, tinggal kita simpen title-nya di heading-nya

Update tampilan **resources > views >posts.blade.php**

```
<h1 class="mb-5">{{ $title }}</h1>
```

Jadi sekarang kita bisa hapus view category **resources >views >category.blade.php**

Sekarang kelihatannya, yang kita lakukan sudah ok semua.

Tapi sebenarnya ada problem **N+1 Problem**

Apa itu N+1 Problem

Source :

<https://www.brentozar.com/archive/2018/07/common-entity-framework-problems-n-1/>

The N + 1 problem occurs when an application gets data from the database, and then loops through the result of that data. That means we call to the database again and again and again. In total, the application will call the database once for every row returned by the first query (N) plus the original query (+ 1).

Masalah N + 1 terjadi saat aplikasi mendapatkan data dari basis data, lalu mengulang hasil data tersebut. Itu berarti kita memanggil basis data berulang kali. Secara total, aplikasi akan memanggil basis data sekali untuk setiap baris yang dikembalikan oleh kueri pertama (N) ditambah kueri awal (+ 1).

Didalam aplikasi kita terjadi looping, dan bahkan butuh tabel lain.

ada 20x ke tabel user + 20x ke tabel category = **40 query**

bayangan klo web-nya udah besar, post-nya udah banyak, penulisnya udah banyak, bisa sampai 200 post, bahkan 2000 post !!!

Problem yang kita hadapi itu dinamakan dengan **N+1 Problem**

By defult di laravelnya menerapkan **lazzy loading** saat terjadi looping.

Loadingnya males, saat dibutuhin baru dilakukan.

Itu bisa jadi bagus, bisa jadi tidak bagus ketika kasusnya seperti ini

Nah, kalo semisal kita pengen ngeliat ben er-bener bahwa yang kita lakukan itu ngga efektif. Kita bisa menggunakan sebuah **library** yang namanya **Clockwork**

Install library Clockwork

Dokumentasi :

<https://github.com/itsgoingd/clockwork>

<https://underground.works/clockwork/>

Library laravel yang akan kita hubungkan ke extension di browser

Untuk mengetahui sebenarnya aplikasi php kita itu melakukan pemanggilan query berapa kali.

Cara install

```
$ composer require itsgoingd/clockwork
```

Install Clockwork jika tidak berhasil tambahkan

```
$ composer install --ignore-platform-reqs
```

Kemudian install extension di browser

Setting > More tools > Extensions > Chrome web store

Clockwork by itsgoingd / underground.works

Kalau sudah berhasil di install, nanti ekstension ini akan muncul di developer tools kita. kalau kita buka halaman yang dibikin pakai php.

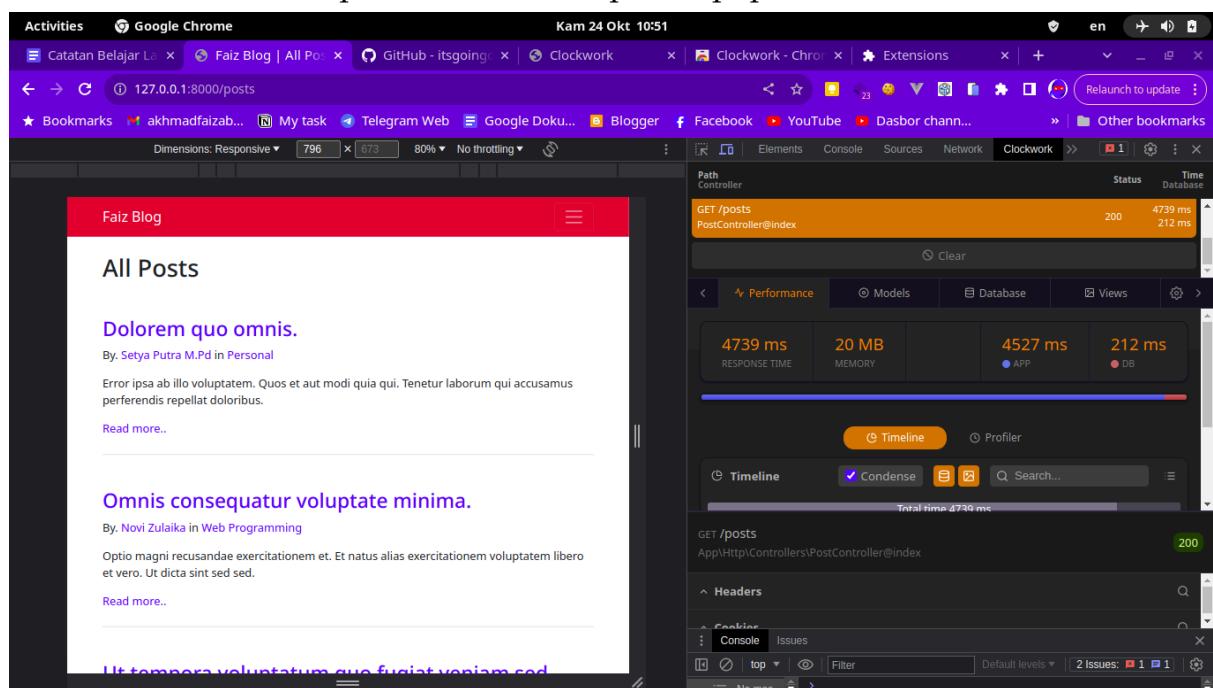
Arahkan ke halaman posts dulu

<http://127.0.0.1:8000/posts>

klik kanan > Inspect >

setelah inspector nya muncul,
di tab baru akan ada menu baru >>> Clockwork

Disini kita bisa melihat performance dari aplikasi php kita



Di menu database bisa kita lihat ada 41 query yang terjadi.

The screenshot shows the Clockwork extension interface. At the top, there's a navigation bar with tabs: Elements, Console, Sources, Network, Clockwork (which is active), and a few others. Below the navigation bar, there's a table with one row. The row contains the URL 'GET /posts' and the controller 'PostController@index'. To the right of the URL are the status '200', time '4739 ms', and database time '212 ms'. Below the table is a 'Clear' button. Underneath the table, there are tabs for Performance, Models, Database (which is active), Views, and a settings gear icon. The Database tab displays performance statistics: 41 QUERIES, 41 SELECTS, and 212 ms TIME. Below these stats is a table titled 'Queries' with columns: Model, Query, and Duration. One entry is shown: 'Post' with the query 'SELECT * FROM `posts` ORDER BY `created_at` DESC' and a duration of '157.73 ms' from 'PostController.php:15'. At the bottom of the interface, there's another table for the same request, showing the URL 'GET /posts', controller 'App\Http\Controllers\PostController@index', and a status of '200'.

Dan ini bisa kita **persingkat**, supaya N+1 nya tidak terjadi.

Kita bisa melakukannya dengan sesuatu yg disebut **Eager loading** (semangat) kebalikan dari **Lazy loading**

Eager Loading

Lazy loading dilakukan ketika dibutuhkan

Eager loading lakukan semua di awal, sehingga kita udah punya datanya.

Sehingga pada **saat looping** nanti tidak perlu melakukan pemanggilan data ke database lagi.

Dokumentasi :

<https://laravel.com/docs/8.x/eloquent-relationships#eager-loading>

Eager loading terjadi ketika kita melakukan relationship, makanya ada di dalam relationship dokumentasinya.

When accessing Eloquent relationships as properties, the related models are "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the " $N + 1$ " query problem. To illustrate the $N + 1$ query problem, consider a Book model that "belongs to" an Author model:

Saat mengakses relasi Eloquent sebagai properti, model terkait "lazy load". Ini berarti data relasi tidak benar-benar dimuat hingga Anda pertama kali mengakses properti. Namun, Eloquent dapat "eager load" relasi saat Anda mengkueri model induk. Eager loading mengurangi masalah kueri " $N + 1$ ". Untuk mengilustrasikan masalah kueri $N + 1$, pertimbangkan model Book yang "termasuk" dalam model Author:

Penjelasan Pak Sandhika :

- * Pada saat kita mengakses sebuah **relationship** di dalam **Eloquent** (setiap kita pakai belongsTo, hasMany, dsb.) maka modelnya akan melakukan sebuah teknik yang namanya **Lazy loading**.
- * Artinya, data relationshipnya ini ngga di load/tidak dipanggil. Sampai nantinya kita **mengakses propertinya** pada saat kita **looping**.
- * Tapi kita bisa minta si Eloquent-nya agar melakukan **Eager loading** ketika kita melakukan query pada parent-nya. Pada saat kita melakukan query di postingannya. Sekalian nge-query langsung author category-nya.
- * **Eager loading** dilakukan untuk menghindari $N+1$ Problem.

Contohnya :

Ada buku dan penulis.

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

Sekarang, mari kita ambil semua buku dan penulisnya:

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

* Problem!

Jadi setiap 25 buku, kita lakukan 26 query
1 query untuk buku , 25 untuk query author-nya

untuk menghindari ini...

Untungnya, kita bisa gunakan **Eager loading** agar mengurangi operasinya hanya jadi **2 query**.

Daripada 26 wkwk.

Cara nya tinggal kita tambahin sebuah **method** yang namanya **with()**
(pak sandhika bacanya wift)

```
$books = Book::with('author')->get();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

* Kalau 2 data kita bisa kasih array

Eager Loading Multiple Relationships

```
$books = Book::with(['author', 'publisher'])->get();
```

Sekarang kita terapkan ke aplikasi kita!

Kita buka controller post kita `http > Controllers > PostController.php`

Kita ubah menjadi seperti ini

```
"posts" => Post::with(['author', 'category'])->latest()->get()
```

* Sekalian ambilin author dan kategori

Sehingga pas looping, dia ngga nge-query. Tapi udah ngambil dari data yang ada di dalam post-nya sekaligus.

* Kita cek web kita lagi dengan Clockwork.

The screenshot shows the Clockwork extension in the Chrome DevTools. The main interface displays a summary of database queries for a request to '/posts'. It shows 3 queries, 3 selects, and a total time of 408 ms. Below this, a detailed table lists two queries: one from the Post model and one from the User model. The Post query is a SELECT * FROM 'posts' ORDER BY 'created_at' DESC, and the User query is a SELECT * FROM 'users' WHERE 'users'.id IN (1, 2, 3, 4, 5). The total duration for these queries is 386.45 ms and 16.73 ms respectively. The interface also includes sections for Headers, Cookies, Console, and Issues.

Model	Query	Duration
Post	SELECT * FROM `posts` ORDER BY `created_at` DESC	PostController.php:15 386.45 ms
User	SELECT * FROM `users` WHERE `users`.`id` IN (1, 2, 3, 4, 5)	PostController.php:15 16.73 ms

Di **menu database** bisa kita lihat web kita hanya melakukan **3 query** saja.

Meskipun nanti kita punya 1000 post, tetap hanya melakukan **3 query** saja.

Performance jauh meningkat!

Jadi saat aplikasi kita lambat, jangan dulu nyalahin database-nya.
jangan dulu nyalahin server -nya.
Cek dulu, mungkin aja kodingan kita kurang optimal.

Lazy Eager Loading

Kita masih punya **2 problem**

Ketika kita masuk halaman author

<http://127.0.0.1:8000/authors/username.author>

Kita cek dengan Clockwork.

The screenshot shows a Google Chrome browser window with three tabs open: 'Catatan Belajar Laravel 8', 'Faiz Blog | Post By Author', and 'Eloquent: Relationships'. The main content area displays a blog post titled 'Post By Author : Cahyadi Mustofa' with two entries: 'Ut eum sapiente.' and 'Placeat est sed omnis.'. The Clockwork extension is active, showing the 'Database' tab with one query listed:

Model	Query	Duration
User	SELECT * FROM `users` WHERE `username` = 'prastuti.chandra' LIMIT 1	3.2 ms HandleCors.php:38

* Sama seperti sebelumnya, di halaman ini melakukan query mengulang sampai banyaknya postingan milik author tersebut.

Disini sampai **14 query**.

Kita bisa mempersingkat juga, namun agak berbeda.

Karena halaman author ini kita pakai route model binding.

kita buka lagi **routes** kita

routes > web.php

```
Route::get('/authors/{author:username}', function (User $author) {  
    return view('posts', [
```

```
        'title' => "Post By Author : $author->name",
        'posts' => $author->posts,
    ]) ;
}) ;
```

* Kalau di PostController, kita tidak pakai route model binding. Jadi gampang aja.

Disini tidak bisa kita tambahin with(), karena yang kita cari adalah author.

```
'posts' => $author->posts,
```

Bagaimana benerinya?

Jadi ini ada cara yang lain.
Ada teknik yang lain.

Jadi kita **tidak pakai** Eager loading.

Tapi kita pakai sesuatu yang disebut dengan **Lazy Eager loading**.
Ini lebih keren lagi!!!

Jadi **digabung**, Lazy loading dengan Eager loading

Dokumentasi :

<https://laravel.com/docs/8.x/eloquent-relationships#lazy-eager-loading>

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

Kadang-kadang kita butuh melakukan Eager loading pada relationship kita, tapi setelah parent- nya sudah didapatkan.

Jadi ngga sekalian di ambil, karena kita lagi melakukan route model binding.

Jadi tidak pakai with(), tapi **pakai load()**.

Jadi pada saat manggil Authornya, kita load juga model sisanya.

Contoh :

```
use App\Models\Book;

$books = Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

Sekarang kita terapkan ke aplikasi kita!

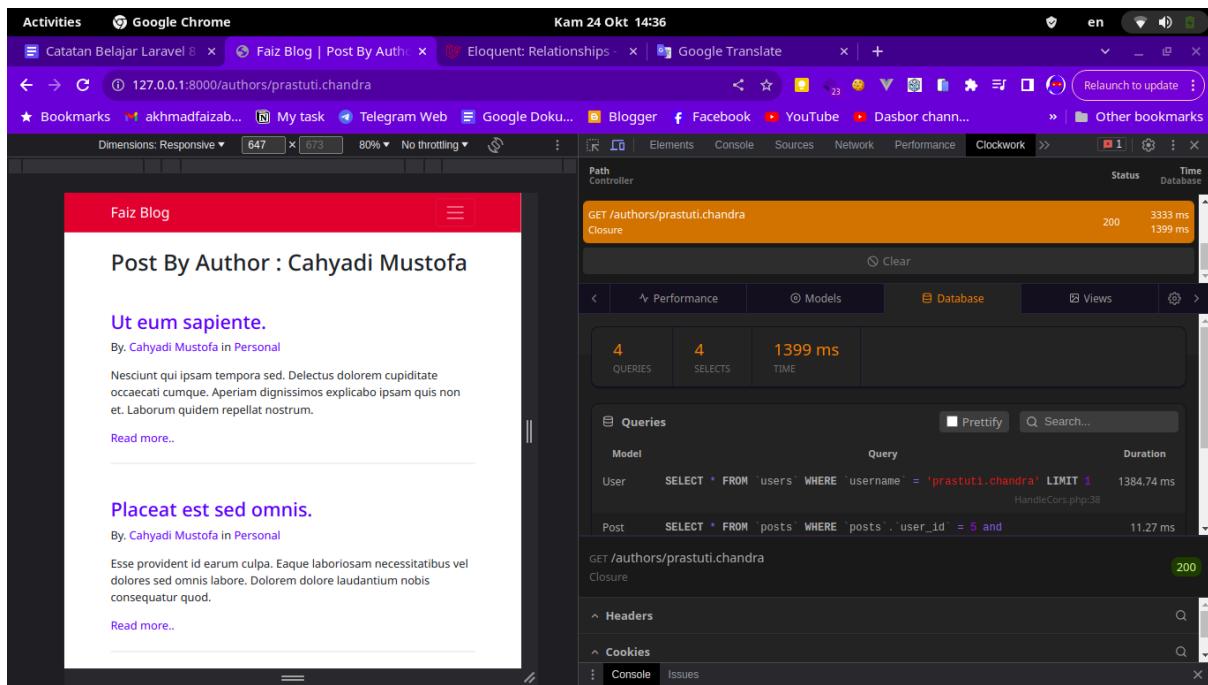
kita buka lagi **routes** kita

routes > web.php

kita tambahkan load() seperti ini di Route Author

```
'posts' => $author->posts->load('category', 'author'),
```

Kita cek lagi **halaman author** dengan Clockwork.



* Sekarang cuma menjalankan **4 query saja**.

Terakhir kita perbaiki halaman kategori

<http://127.0.0.1:8000/categories/web-programming>

kita tambahkan load() juga seperti ini di Route Category

```
Route::get('/categories/{category:slug}', function (Category $category) {
    return view('posts', [
        'title' => "Post By Category : $category->name",
        'posts' => $category->posts->load('category', 'author')
    ]);
});
```

* Meskipun kebalik antara `load('category', 'author')`
atau `load('author', 'category')`
tidak masalah!

Kita cek juga halaman kategori dengan Clockwork.
Dan web kita sudah jauh lebih optimal dari sebelumnya.

12. Redesign UI

Memperbaiki halaman posts

Kita buka view Posts **resources > views >posts.blade.php**

kita buat agar halaman posts menampilkan yang pertama ada hero post-nya.
Post yang terbaru akan muncul sebagai hero-nya (jadi besar sendiri)
dibawahnya (post-post yang lain akan muncul sebagai card kecil-kecil)
1 baris 3 buah card

Kita buka dulu dokumentasi Bootstrap-nya

<https://getbootstrap.com/docs/5.0/getting-started/introduction/>

Kita copy-kan yang kita butuhkan.

untuk Hero post

<https://getbootstrap.com/docs/5.0/components/card/#images-1>

Sebelum itu, kita akan cek postingannya ada atau tidak.

Mungkin aja kosong, Jadi kita kasih kondisi.

Kita akan kita hitung jumlah dari postingannya. menggunakan **@if()** punya-nya blade

```
@if ($posts->count())
    {{-- kondisi jika true --}}
@else
    {{-- kondisi jika false --}}
    <p>No post found.</p>
@endif
```

* method count() aka menghitung jumlah dari postingannya

Sebenarnya kita bisa tulis lebih besar dari 0 seperti ini.

```
@if ($posts->count() > 0)
```

Namun meskipun kita hapus lebih besar dari 0-nya pun akan tetap menghasilkan true jika nilainya lebih dari 0, dan akan menghasilkan false jika kosong.

Sekarang kita pengen agar yang tampil di hero post adalah postingan yang terakhir.

Kalau kita lihat di post controller kita (**http > Controllers > PostController.php**), kita punya method latest(), jadi kita sudah mencari yang postingan paling akhir.

Memindahkan eager loading ke model

Kita perbaiki agar tidak perlu mengetikkan with() di PostController kita. Kita bisa simpan with() di model post.

post controller (**http > Controllers > PostController.php**)

```
"posts" => Post::with(['author', 'category'])->latest()->get()
```

Kita bisa kasih tau agar setiap pemanggilan data Post:: di PostController kita, dia sudah membawa author dan category.

Jadi with() nya bisa kita simpan di model post. tidak didalam controller

Caranya kita tinggal **tambahkan properti** baru di model **app > Models > Post.php**

```
protected $guarded = ['id'];
protected $with = ['category', 'author'];
```

Jadi di post controller cukup gini aja.

http > Controllers > PostController.php

```
"posts" => Post::latest()->get()
```

* kita cek halaman posts dan author kita di web-nya

Membuat hero posts

kita punya `latest()` , jadi kalau kita panggil postingan pertama itu pasti postingan terbaru.

Jadi ketika masuk kedalam collection,

```
"posts" => Post::latest()->get()
```

kita ambil index yang ke-0

Update view Posts `resources > views >posts.blade.php`

```
@if ($posts->count() > 0)
    {{-- kondisi jika true --}}
    <div class="card mb-3">
        
        <div class="card-body text-center">
            <h3 class="card-title">{{ $posts[0]->title }}</h3>
            <p>
                <small class="text-muted">
                    By. <a href="/authors/{{ $posts[0]->author->username
}}" class="text-decoration-none">{{ $posts[0]->author->name }}</a> in <a
href="/categories/{{ $posts[0]->category->slug }}">
                <small>
                    {{ $posts[0]->category->name }}</a>
                    {{ $posts[0]->created_at->diffForHumans()  }}
                </small>
            </p>
            <p class="card-text">{{ $posts[0]->excerpt }}</p>
            <a href="/posts/{{ $posts[0]->slug }}">
                <small>Read more</small>
            </a>
        </div>
    </div>
@else
    {{-- kondisi jika false --}}
    <p class="text-center fs-4">No post found.</p>
@endif
```

* Mengganti kata/variabel yang sama sekaligus di vscode dengan **Ctrl + D** (4x atau sampai kata/variabel yang sama yang akan diganti)

Bagaimana kita mau menampilkan kapan postingan di buat?

Jadi kita punya data **created_at** dimana berupa tanggal. agar mudah dibaca (semisal : 3 menit yang lalu) kita bisa pakai **library** yang namanya **carbon** (untuk mengatur waktu), dan sudah ada di dalam laravel.

```
{ { $posts[0]->created_at->diffForHumans() } }
```

Menambahkan post image (unsplash api)

<https://source.unsplash.com/>

Unsplash API

<https://source.unsplash.com/1600x900/?nature,water>

Additional :

Unsplash API Updated with key access

https://api.unsplash.com/photos/random?client_id=PyZzkf1DT0ZlmS6DbpMkUdk92gwSfWHWV_e7xCKF48

```
<script>
    const imageElement =
document.getElementById('random-image');
```

```
        const apiKey = `{{ env('UNSPLASH_ACCESS_KEY') }}`; //  
Ambil API key dari .env  
  
        function getRandomImage() {  
  
fetch(`https://api.unsplash.com/photos/random?client_id=${apiKey}`)  
        .then(response => response.json())  
        .then(data => {  
            imageElement.src = data.urls.regular;  
        })  
        .catch(error => {  
            console.error('Error fetching image:', error);  
        });  
    }  
  
    // Panggil fungsi saat halaman dimuat dan setiap 5 detik  
    window.onload = getRandomImage;  
    setInterval(getRandomImage, 5000);  
</script>
```

```
<script>  
    let cache = {}; // Simpan URL gambar yang sudah didapatkan  
  
    function getRandomImage(width, height) {  
        const key = `${width}x${height}`;  
  
        if (cache[key]) {  
            imageElement.src = cache[key];  
            return;  
        }  
  
        axios.get(`https://api.unsplash.com/photos/random?client_id=${apiKey}&w=${1200}&h=${400}`)  
        .then(response => {  
            imageElement.src = response.data.urls.regular;  
            cache[key] = response.data.urls.regular;  
        })  
        .catch(error => {  
            console.error('Error fetching image:', error);  
        });  
    }  
</script>
```

```
    });
}
</script>
```

<https://images.unsplash.com/photo-1461988320302-91bde64fc8e4?ixid=2yJhcHBfaWQiOjEyMDd9&w=1500&dpr=2>

https://api.unsplash.com/photos/random?client_id=PyZzkf1DT0ZlmS6DbpMkUdk92gwSfWHWV_e7xCKF48

<https://images.unsplash.com/photo-1461988320302-91bde64fc8e4?ixid=2yJhcHBfaWQiOjEyMDd9&w=1500&dpr=2>

<https://ix-www.imgur.com/case-study/unsplash/unplash03.jpg?ixlib=js-3.8.0&auto=compress%2Cformat&w=1446>

Cara Memperbaiki Gambar Gepeng atau Melebar Dengan CSS

<https://www.mediaterbuka.com/2020/10/cara-memperbaiki-gambar-gepeng-atau-melebar-dengan-css.html>

Gambar gepeng, jadi melebar, tidak bagus.

<https://app.sko.dev/post/gambar-gepeng-jadi-melebar-tidak-bagus-1568820988>

5 Ways to Crop Images in HTML/CSS

<https://cloudinary.com/guides/automatic-image-cropping/5-ways-to-crop-images-in-html-css>

```
>
```

```
<h3 class="card-title">
    <a href="/posts/{{ $posts[0]->slug }}">
        {{ $posts[0]->title }}</a>
    </h3>
```

Membuat card untuk post

kita buat foreach-nya ngulang semua kecuali postingan yang pertama
kita bisa memberi sebuah **method** yang namanya **skip()**

```
@foreach ($posts->skip(1) as $post)
```

Kita bikin struktur halamannya,
kita akan bikin card yang jejer ada 3
berarti artinya kita butuh **kolom** yg masing-masih ukurannya 4
karena total semua harus 12.

```
<div class="container">
    <div class="row">
        <div class="col-md-4">

        </div>
    </div>
</div>
```

Kita copy-kan card dari bootstrap
<https://getbootstrap.com/docs/5.0/components/card/#example>

```

<div class="container">
    <div class="row">

        @foreach ($posts->skip(1) as $post)

            <div class="col-md-4 mb-3">
                <div class="card">
                    category->name }}">
                    <div class="card-body">
                        <h5 class="card-title">{{ $post->title }}</h5>
                        <p>
                            <small class="text-muted">
                                By. <a href="/authors/{{ $post->author->username }}" class="text-decoration-none">{{ $post->author->name }}</a>
                            <small>
                                {{ $post->created_at->diffForHumans() }}
                            </small>
                        </p>
                        <p class="card-text">{{ $post->excerpt }}</p>
                        <a href="/posts/{{ $post->slug }}" class="btn btn-primary">Read more</a>
                    </div>
                </div>
            </div>

        @endforeach

    </div>
</div>

```

Kemudian kita hapus foreach yang sebelumnya.

Menambahkan link ke post category

Tambahkan di dalam card

```
<div class="card">
    <div class="position-absolute px-3 py-2 text-white"
style="background-color:rgba(0, 0, 0, 0.7)">
        <a href="/categories/{{ $post->category->slug }}"
class="text-white text-decoration-none">
            {{ $post->category->name }}
        </a>
    </div>
</div>
```

Memperbaiki halaman single post

Kita buka view single Post **resources > views >posts.blade.php**

```
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">

            <h1 class="mb-5">{{ $post->title }}</h1>

            <p>By. <a href="/authors/{{ $post->author->username }}" class="text-decoration-none">{{ $post->author->name }}</a> in <a href="/categories/{{ $post->category->slug }}" class="text-decoration-none">{{ $post->category->name }}</a></p>

            {!! $post->body !!}
```

```
<a href="/posts" class="d-block mt-3">Back To  
Posts </a>  
  
    </div>  
  </div>  
</div>
```

Memperbaiki seed dan factory agar paragraf yang dibuat di bungkus dengan tag `<p></p>`

Dokumentasi :

<https://fakerphp.org/formatters/text-and-paragraphs/#paragraph>

<https://fakerphp.org/formatters/text-and-paragraphs/#paragraphs>

Kalau paragraphs (yang ada s nya), dia akan membuat sebuah array yang didalamnya beberapa paragraf.

Kita update `postFactory.php` kita

```
'body' => '<p>' .  
implode('</p><p>', $this->faker->paragraphs(mt_rand(5,10))) . '</p>' ,
```

* `implode()` itu di join sama kayak di javascript.
setiap array-nya dengan menambahkan delimiter-nya/pemisahnya adalah `</p><p>`

Atau kita bisa pakai map

```
'body' => collect($this->faker->paragraphs(mt_rand(5,10)))  
          ->map(function($p) {  
            return "<p>$p</p>" ;  
          }) ,
```

* Jadi kita bungkus dulu si paragraf ke dalam sebuah collection.
`collect()`, membuat array biasa menjadi collection.

Kemudian kita map().

kita petakan setiap elemen array-nya agar nanti di tempelin paragraf.

Setiap paragraf-nya kita ganti jadi \$p, kita closure function(\$p) .

kita map agar dia mengembalikan (return) <p>\$p</p>

Atau mau kita sederhanakan menjadi Arrow function (syntax PHP yang baru) menjadi seperti ini

```
'body' => collect($this->faker->paragraphs(mt_rand(5,10)))
          ->map(fn($p) => "<p>$p</p>") ,
```

Kita cek file **DatabaseSeeder** kita lagi sebelum melakukan migrasi

Kemudian kita migrate fresh pake seed-nya juga

```
$ php artisan migrate:fresh --seed
```

Maka, datanya akan seperti ini... Taraaaa...

["

Dolor et atque iusto pariatur numquam inventore. Cum nam omnis tenetur aut fuga
et eum. Laudantium nemo dolore eum consequatur enim recusandae sit
necessitatibus.<\p>","

Sed blanditiis et magni perferendis impedit qui. Esse cumque quas commodi
molestiae. Cupiditate et saepe repellendus necessitatibus. Velit aut delectus iste
dolorem.<\p>","

* bentuknya masih array, belum di join.

Kita perbaiki lagi **postFactory.php** kita dengan menambahkan implode()

```
'body' => collect($this->faker->paragraphs(mt_rand(5,10)))
          ->map(fn($p) => "<p>$p</p>")
          ->implode(' ') ,
```

* kita implode() dengan delimiter-nya/pemisahnya adalah **kutip** (‘ ’)

Kemudian kita migrate fresh pake seed-nya juga

```
$ php artisan migrate:fresh --seed
```

Setelah itu kita cek data postingan di database kita atau di web kita.

* class img-fluid supaya responsive

Sekarang baru kita perbaiki tampilan single post kita.

Kita update view single Post **resources > views >posts.blade.php**

```
<div class="container">
    <div class="row justify-content-center mb-5">
        <div class="col-md-8">

            <h1 class="mb-3">{{ $post->title }}</h1>

            <p>By. <a href="/authors/{{ $post->author->username }}" class="text-decoration-none">{{ $post->author->name }}</a> in <a href="/categories/{{ $post->category->slug }}" class="text-decoration-none">{{ $post->category->name }}</a></p>

            category->name }}">

            <article class="my-3 fs-5">
                {!! $post->body !!}
            </article>
```

```
        <a href="/posts" class="d-block mt-3">Back To  
Posts </a>  
  
        </div>  
    </div>  
    </div>
```

Memperbaiki halaman categories

Sekarang kita akan memperbaiki halaman **/categories** kita

<http://127.0.0.1:8000/categories>

Kita perbaiki navbar kita dulu.

Update file navbar kita

resources > views > partials > navbar.blade.php

bikin satu lagi untuk categories

```
<li class="nav-item">
    <a class="nav-link {{ ($title === "Categories") ? 'active' : '' }}" href="/categories">Categories</a>
</li>
```

kita perbaiki active-nya juga.

Kita ubah semua title menjadi variabel active

```
<li class="nav-item">
    <a class="nav-link {{ ($active === "home") ? 'active' : '' }}" href="/">Home</a>
</li>
<li class="nav-item">
    <a class="nav-link {{ ($active === "about") ? 'active' : '' }}" href="/about">About</a>
</li>
<li class="nav-item">
    <a class="nav-link {{ ($active === "posts") ? 'active' : '' }}" href="/posts">Blog</a>
</li>
<li class="nav-item">
    <a class="nav-link {{ ($active === "categories") ? 'active' : '' }}" href="/categories">Categories</a>
</li>
```

* Ini merupakan cara dari pak sandhika
Ada juga cara yang mungkin lebih efektif

Bagaimana cara mengaksesnya?
kita bisa buka controller kita **http > Controllers > PostController.php**

Jadi controller selain ngirimin title, kita ngirimin active juga.

```
public function index()
{
    return view('posts', [
        "title" => "All Posts",
        "active" => 'posts',
        // "posts" => Post::all()
        "posts" => Post::latest()->get()
    ]);
}

public function show(Post $post)
{
    return view('post', [
        "title" => "Single Post",
        "active" => 'posts',
        "post" => $post
    ]);
}
```

Terus yang lain ada di routes kita,
kita buka file **routes > web.php**

```
Route::get('/categories', function() {
    return view('categories', [
        'title' => 'Post Categories',
        'categories' => Category::all()
    ]);
});
```

Kita percantik tampilan halaman/**categories** kita
<http://127.0.0.1:8000/categories>

Bootstraps :

<https://getbootstrap.com/docs/5.0/components/card/#image-overlays>

```
<div class="container">
    <div class="row">

        @foreach ($categories as $category)

            <div class="col-md-4">
                <a href="/categories/{{ $category->slug }}">
                    <div class="card bg-dark text-white">
                        name }}">
                        <div class="card-img-overlay d-flex
align-items-center p-0">
                            <h5 class="card-title text-center
flex-fill p-4 fs-3" style="background-color: rgba(0, 0, 0, 0.7)">{{
$category->name }}</h5>
                        </div>
                    </div>
                </a>
            </div>

        @endforeach

    </div>
</div>
```

13. Searching & Pagination

Membuat form searching

Kita buka view Posts **resources > views >posts.blade.php**

Kita pindahkan else-nya ke bawah container.

Supaya ketika postingannya tidak ada pada saat kita cari, ini yang akan tampil.

```
@else
    {{-- kondisi jika false --}}
    <p class="text-center fs-4">No post found.</p>
@endif
```

Kita buat form penyimpanannya.

```
<div class="row">
    <div class="col-md-6">
        <form action="/posts" method="GET"></form>
    </div>
</div>
```

* method-nya GET, dan meskipun tidak di tuliskan method **default-nya GET**

Kita copy form-nya dari bootstraps :

<https://getbootstrap.com/docs/5.0/forms/input-group/#button-addons>

Tampilannya udah OKE, tinggal kita jalankan pencarinya.

Kita cari tahu dulu, gimana cara menangkap apa yang kita tulis di input search.

Jadi pada saat kita pencet search, akan dikirimkan ke halaman posts, name-nya search yang isinya “keyword pencarian”

<http://127.0.0.1:8000/posts?search=keyword+pencarian>

Nah, gimana cara nangkep ini?? kita bisa pakai **method REQUEST**

Menambahkan query searching

Caranya..

Karena kita tahu dia dikirim ke halaman posts, jadi kita harus arahkan dulu ke routes-nya

Kita punya PostController

Coba kita tangkap dulu dengan **dd(request('search'));**

```
public function index()
{
    dd(request('search'));

    return view('posts', [
        "title" => "All Posts",
        "active" => 'posts',
        // "posts" => Post::all()
        "posts" => Post::latest()->get()
    ]);
}
```

Kemudian kita coba search.

Tinggal kita lakukan QUERY
kita akan pecah query-nya.
disini kita mencari sesuai judul.

```
public function index()
{
    // dd(request('search')); //testing nangkep input search

    $posts = Post::latest();
}
```

```

        if (request('search')) {
            $posts->where('title', 'like', '%' . request('search')
. '%');
        }

        return view('posts', [
            "title" => "All Posts",
            "active" => 'posts',
            // "posts" => Post::all()
            // "posts" => Post::latest()->get()
            "posts" => $posts->get()
        ]);
    }
}

```

* %search% supaya dia bisa mencari apapun yang ada di depannya dan apapun yang ada di belakangnya.

Kita tambahkan value di form input search `value="{{ request('search') }}`"> agar saat search kita tahu apa yang kita search.

```

<input type="text" class="form-control" placeholder="Search.."
name="search" value="{{ request('search') }}">

```

Semisal kita pengen cari juga, selain dari judul.
Kita pengen cari dari apapun yang ada di body-nya

kita tinggal tambahkan OR

```

        if (request('search')) {
            $posts->where('title', 'like', '%' . request('search')
. '%')
                ->orWhere('body', 'like', '%' . 
request('search') . '%');
        }
}

```

Namun saat kita melakukan pencarian data seperti itu, kemungkinan besar adalah **tugasnya MODEL**, bukan tugasnya controller walaupun jalan normal.

Query Scope

Dokumentasi :

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

Fitur dari laravel (spesifiknya fitur dari eloquent) agar kita bisa membuat filter kita sendiri.

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with scope.

Dengan menggunakan **Local scopes**, itu memungkinkan kita untuk mendefinisikan query-query umum yang bisa kita gunakan kembali di dalam aplikasi kita.

Contohnya kalo misalkan kita butuh untuk secara sering mengambil data user yang cukup populer

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     *

```

```

 * Scope a query to only include popular users.
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @return \Illuminate\Database\Eloquent\Builder
 */
public function scopePopular($query)
{
    return $query->where('votes', '>', 100);
}

/**
 * Scope a query to only include active users.
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @return void
 */
public function scopeActive($query)
{
    $query->where('active', 1);
}

```

* Kalau pengen cari caranya...

Di Dalam model kita bikin yang namanya scope.

Jadi kita tulis sebuah **method** yang namanya **scopePopular(\$query)**.

kata scope wajib, dan kata setelahnya scope-nya bebas.

Tapi di depannya harus ada scope-nya, buat ngasih tau bahwa dia adalah query scope.

terus di dalamnya tinggal kita tulis **klausul tambahannya** apa

```
return $query->where('votes', '>', 100);
```

Misalkan didepan kita udah cari,

```
select * from user
```

nah kalau mau cari yang populer, kita tinggal tambahin sisipkan **where()**. misalkan kita mau cari user yang nge-votes lebih dari 100

Dan ini persis dengan punya kita.

Kita cari post di awal (semua post), lalu nanti kita sisipkan filternya. misalnya searching, jadi namanya boleh **scopeSearching(\$query)** .

Karena nanti kita mau bikinnya jadi agak general (generic) kita nanti tulisnya **scopeFilter(\$query)** , karena filternya bisa banyak. tinggal nanti kita return query yang mau kita lakukan. dalam hal ini **WHERE LIKE**

Kalau sudah kita pindahin ke dalam model, cara manggilnya yaitu kita bisa panggil langsung di dalam method-nya.

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Sekarang kita cobain!

Kita update model post kita **app > Models > Post.php**

```
public function scopeFilter($query)
{
    if (request('search')) {
        return $query->where('title', 'like', '%' . request('search') .
        '%')
            ->orWhere('body', 'like', '%' . request('search') .
        '%');
    }
}
```

Kemudian kita kembalikan controller kita menjadi seperti ini **http > Controllers > PostController.php**

```
public function index()
{
    return view('posts', [
        "title" => "All Posts",
        "active" => 'posts',
        "posts" => Post::latest()->filter()->get()
    ]);
}
```

```
    ];
}
```

Kemudian **kita coba search** postingan (judul atau isi) yang ingin kita cari di web kita.

Reuest('search') harusnya dilakukan di controller

Sekarang codingan kita udah tambah rapi.
Kalau mau di rapihin lagi, **bisa!**

Karena **request('search')** dilakukan di model.

Kalau **data** itu kerjaannya **model**

```
return $query->where('title', 'like', '%' . request('search') . '%')
    ->orWhere('body', 'like', '%' . request('search') . '%');
```

Kalau **request('search')** sebenarnya kerjaan-nya **controller**. untuk ngecek ada request apa ngga..
bukan kerjaan-nya model

request('search') akan kita tarik ke **controller**.

Jadi nanti caranya kita cari, ada ngga **variable search** yang dikirim ke method scopeFilter ini

jadi semisal nanti kita kirim dalam bentuk **array** dengan **variabel \$filters** (karena akan kita pakai untuk pencarian yang lebih kompleks)
kita akan cek menggunakan isset (jadi agak panjang).

Kita update model post kita **app > Models > Post.php**

```
public function scopeFilter($query, array $filters)
{
    if (isset($filters['search']) ? $filters['search'] : false) {
        return $query->where('title', 'like', '%' . $filters['search'] .
    '%' )
```

```
        ->orWhere('body', 'like', '%' . $filters['search'] . '%');
    }
}
```

* pakai ternary operator ?:

```
isset($filters['search']) ? $filters['search'] : false
```

Kemudian kita update juga controller kita menjadi seperti ini **http > Controllers > PostController.php**

```
return view('posts', [
    "title" => "All Posts",
    "active" => 'posts',
    "posts" => Post::latest()->filter(request(['search']))->get()
]);
```

Kemudian **kita coba search** postingan (judul atau isi) yang ingin kita cari di web kita. Dan hasilnya sama saja.

Mengganti IF dengan Method when()

Sekarang akan kita bikin menjadi lebih efisien lagi.

Kita akan **ganti if** nya menggunakan method di laravel yang namanya **when()**

Dokumentasi :

<https://laravel.com/docs/8.x/collections#method-when>

when method akan dijalankan ketika first argument berisi true

```
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});
```

```
$collection->when(false, function ($collection) {
    return $collection->push(5);
}) ;

$collection->all();

// [1, 2, 3, 4]
```

* Jadi kita ngga usah pakai IF lagi, tapi nanti kita lihat dari collection yang kita punya.

Kita lakukan ini, karena nanti kita akan bikin IF nya banyak. Kita ganti menjadi when().

Sebetulnya hanya mengganti notasi saja, fungsinya tetap sama.

Tapi nanti akan lebih ringkas kalau kita udah punya banyak

Null coalescing operator php

Nah kita bisa meringkas

Kalau pakai PHP 7 , ada yang namanya null coalescing operator

Dokumentasi PHP :

<https://www.php.net/manual/en/migration70.new-features.php#migration70.new-features.null-coalesce-op>

The null coalescing operator (??) has been added as syntactic sugar for the common case of needing to use a ternary in conjunction with `isset()`. It returns its first operand if it exists and is not `null`; otherwise it returns its second operand.

Null coalescing operator yaitu (??) sudah ditambahin di php 7 supaya menambahkan **syntactic sugar** (pemanis) supaya kita nulisnya gampang saat kita butuh untuk menggunakan ternary dan juga digunakan untuk ngecek `isset()`

```

<?php
// Fetches the value of $_GET['user'] and returns 'nobody'
// if it does not exist.
$username = $_GET['user'] ?? 'nobody';
// This is equivalent to:
$username = isset($_GET['user']) ? $_GET['user'] : 'nobody';

// Coalescing can be chained: this will return the first
// defined value out of $_GET['user'], $_POST['user'], and
// 'nobody'.
$username = $_GET['user'] ?? $_POST['user'] ?? 'nobody';
?>

```

Jika kita terapkan !!!
Yang awalnya seperti ini

```

if (isset($filters['search']) ? $filters['search'] : false) {

```

Menjadi simple seperti ini

```

$query->when($filters['search'] ?? false)

```

* tanpa isset() dan ?:

Sekarang model post kita menjadi seperti ini **app > Models > Post.php**

```

$query->when($filters['search'] ?? false, function($query, $search) {
    return $query->where('title', 'like', '%' . $search . '%')
        ->orWhere('body', 'like', '%' . $search . '%');
});

```

Kemudian kita coba search postingan (judul atau isi) yang ingin kita cari di web kita. Dan hasilnya sama saja. namun codingan kita menjadi lebih rapi lagi.

Query Filter

Kenapa harus ribet-ribet? padahal enak yang pertama tadi.

Nah kalo semisal kita masuk ke halaman kategori/author,

harusnya kita saat mencari postingan tertentu di kategori “Web programming”, harusnya pencarinya spesifik dan yang muncul hanya postingan yang kategori-nya “Web programming” saja.

Caranya, kita bisa manfaatin filter kita tadi.

Kita update controller kita **http > Controllers > PostController.php**

```
"posts" => Post::latest()->filter(request(['search', 'category']))->get()
```

Kita update model post kita **app > Models > Post.php**

Kita tambahkan di bawahnya, atau sebenarnya kita bisa chain-in kasih when lagi di akhir

```
$query->when($filters['search'] ?? false, function($query, $search) {
    return $query->where('title', 'like', '%' . $search . '%')
        ->orWhere('body', 'like', '%' . $search . '%');
})->when();
```

Tapi biar lebih gampang dibaca, kita tambahkan di bawahnya aja.

```
$query->when($filters['category'] ?? false, function($query, $category) {
    return $query->whereHas('category', function($query) use
    ($category) {
        $query->where('slug', $category);
    });
});
```

* disini agak sedikit kompleks karena kita return melakukan join table. kita join tabel kategori.

Jadi kita cari postingan yang sesuai dengan kriteria yang dicari, tapi juga merupakan bagian dari kategori.

Disini laravel sudah punya sebuah method yang namanya **whereHas()**

* Variabel \$category yang di dalam tidak bisa menggunakan variabel \$category yang di luar. maka dari itu kita kirimkan dulu menggunakan **use (\$category)**

Kita cek di web kita

Yang kita punya sebelumnya, tidak pakai request kalau mau mencari kategori. Yang kita punya, kita pakai method di dalam controller atau didalam routes

Mengarah ke /categories/

<http://127.0.0.1:8000/categories/web-programming>

Nah tapi kita udah punya query yang baru.
QUERY kita yang baru itu gini nulisnya.

<http://127.0.0.1:8000/posts?category=web-programming>

* dari halaman posts, kita kasih QUERY-nya atau request variabel-nya (?) adalah **category=**

dengan begitu, hasilnya sama menampilkan semua postingan yang kategorinya web programming. Namun judul halamannya masih All Posts

Nah, kerennya kalau kita udah bisa pakai method yang category, kita bisa gabungin dengan method yang search
caranya tinggal menambahkan **&search=**

<http://127.0.0.1:8000/posts?category=web-programming&search=Nesciunt%20eum%20debitis%20et>

* Kita cek web kita lagi dengan Clockwork.

QUERY-nya sudah sesuai

The screenshot shows the Clockwork extension interface. At the top, there's a navigation bar with tabs: Elements, Console, Sources, Network, Performance, Clockwork (which is active), and others. Below the navigation bar, there's a status bar with 'Path' and 'Controller'. The main area displays a network request: 'GET /posts?category=web-programming&search=Nesciunt%20eum%20debitis%20et' from 'PostController@index'. The status shows '200' and '498 ms' total time, with '9 ms' for the database. Below the request, there are tabs for Performance, Models, Database (which is selected), Views, and Settings. Under the Database tab, it shows '3 QUERIES' and '3 SELECTS' with a total '9 ms' duration. The 'Queries' section lists a single query for the Post model:

Model	Query	Duration
Post	<pre>SELECT * FROM `posts` WHERE (`title` like '%Nesciunt eum debitis et%' or `body` like '%Nesciunt eum debitis et%' and EXISTS (SELECT * FROM `categories` WHERE `posts`.`category_id` = `categories`.`id` and `slug` = 'web-programming')) ORDER BY `created_at` DESC</pre>	7.82 ms

The query is attributed to 'PostController.php:28'. Below the queries, there's a 'Headers' section and a 'GET data' section, both with expand/collapse arrows. At the bottom, there are tabs for 'Console' and 'Issues'.

Namun kalau kita coba search di **form input search-nya**, dia tetap menggunakan link yang tanpa variabel \$category

<http://127.0.0.1:8000/posts?search=katayangdicari>

Nah untuk mengimplementasi, kita harus mengganti semua link-nya agar ngga pakai lagi link ke kategori

<http://127.0.0.1:8000/categories/>

Dan juga kita harus kirimkan data kategori di URL-nya.
Kita bisa titipkan di form pencarian-nya.

Kita ganti link ke kategori yang ada di view posts dan juga single post
resources > views >**posts.blade.php** atau **post.blade.php**

```
<a href="/posts?category={{ $posts[0]->category->slug }}"
```

Kita sisipkan di form pencarian, tambahkan IF

```
@if (request('category'))
    <input type="hidden" name="category" value="{{ request('category') }}"
} }">
@endif
```

Latihan Memperbaiki Pencarian Untuk Author

Sama seperti category, namun di model tinggal mengganti slug dengan username.
dan perlu menambahkan kondisi IF lagi di form pencarian.

Pak Sandhika mencontohkan menggunakan **arrow function**

```
$query->when($filters['author'] ?? false, fn($query, $author) =>
    $query->whereHas('author', fn($query) =>
        $query->where('username', $author)
    )
);
```

* enaknya juga ga perlu lagi pakai **use (\$author)**, karena scope-nya dia udah langsung nyari ke atasnya.

Pak Sandhika mencontohkan menambahkan IF lagi dibawanya

```
@if (request('category'))
    <input type="hidden" name="category" value="{{ request('category') }}">
@endif

@if (request('author'))
    <input type="hidden" name="author" value="{{ request('author') }}">
@endif
```

Kita juga bisa mencari berdasarkan ketiga-nya, namun tidak ada use case seperti itu di web kita kali ini.

<http://127.0.0.1:8000/posts?search=katayangdicari&author=nama-author&category=web-design>

dan jika kita cek di ClockWork pasti QUERY-nya lumayan panjang.

Memperbaiki Halaman Posts

Kita ganti juga link ke kategori yang ada di view categories
resources > views >categories.blade.php

```
<a href="/posts?category={{ $posts[0]->category->slug }}"
```

Sekarang kita perbaiki agar judul halaman posts tidak hanya menampilkan All Posts, tapi mengikuti sesuai request

Kita akalin controller kita menjadi seperti ini
http > Controllers > PostController.php

```
$title = '';
if(request('category')) {
```

```
$category = Category::firstWhere('slug',  
request('category'));  
    $title = ' in ' . $category->name;  
}  
  
if(request('author')) {  
    $author = User::firstWhere('username', request('author'));  
    $title = ' by ' . $author->name;  
}  
  
return view('posts', [  
    "title" => "All Posts" . $title,  
    "active" => 'posts',  
    "posts" => Post::latest()->filter(request(['search',  
'category', 'author']))->get()  
]);
```

Kemudian sekarang kita cek di web kita.

Routes ([web.php](#)) yang sudah tidak di pakai boleh di hapus saja.

Membuat Fitur Pagination

Kita akan menambahkan 1 fitur lagi, yaitu pagination.
Dan harusnya bikin-nya kompleks dan bikinnya lama

Tapi laravel sudah memudahkan untuk membuat ini, dan benar-benar **MAGIC!**
Fitur pagination ini **SANGAT AJAIB** menurut pak Sandhika

Kita bikin postingan yang tampil hanya 7

Jika pernah mengikuti playlist PHP dasar, **bikin-nya lumayan susah.**
Tapi kalau di laravel gampang.

Dokumentasi :

<https://laravel.com/docs/8.x/pagination#basic-usage>

Cara pakai-nya kita cukup panggil **paginate()**

Sebelumnya kita manggil-nya pakai **get()**

Kita akan ganti GET-nya dengan **paginate()**

Dan kita tinggal masukin berapa item per-halaman, semisal 15
paginate(15)

pak Sandhika said :

UDAH, INI NGGA ADA YANG LEBIH AJAIB DARIINI TEMEN-TEMEN


Kita ganti **get()** menjadi **paginate(7)** di Post controller kita **http > Controllers > PostController.php**

```
// "posts" => Post::latest()->filter(request(['search', 'category', 'author']))->get()
"posts" => Post::latest()->filter(request(['search', 'category', 'author']))->paginate(7)
```

Sekarang jika kita cek di web kita, otomatis hanya 7 postingan yang akan ditampilkan.

Kita belum ada halamannya, namun jika kita tambahkan ?page=2

<http://127.0.0.1:8000/posts?page=2>

kita sudah masuk ke halaman ke-2

Gimana cara-nya nambahin link-nya? **LEBIH GAMPANG LAGI !!!**

Kita tinggal nambahin **links()** di view posts kita
resources > views > posts.blade.php

```
{{ $posts->links() }}
```

Sekarang jika kita cek di web kita, akan ada tombol Previews dan Next untuk menampilkan post lainnya.

Tapi kenapa tampilannya berantakan??

Karena defaultnya Laravel menggunakan **TAILWIND**

Nah kita kan pakai **bootstrap..**

gimana dong pak kalau mau pakai bootstrap?

GAMPANG AJA, TINGGAL GANTI AJA

caranya gimana?

KITA COBA GOOGLING !!!

Dokumentasi :

<https://laravel.com/docs/8.x/pagination#using-bootstrap>

Cara pakai bootstrap kita tinggal cari file ini

App\Providers\AppServiceProvider

tinggal nambahin 1 baris di dalam function boot() -nya

```
use Illuminate\Pagination\Paginator;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Paginator::useBootstrap();
}
```

* Tambahkan namespace Paginator-nya.

Kalau pakai tailwind, ga usah.

Sekarang jika kita cek di web kita, tampilan paginator-nya sudah pakai bootstrap.

Namun akan kita percantik lagi dengan mengakali seperti ini

```
<div class="d-flex justify-content-end">
    {{ $posts->links() }}
</div>
```

Tapi ada problem sedikit saat kita melakukan pencarian. Paginationnya bakalan ga jalan.

Cara solve-nya, **GAMPANG BGT !!!**

Kita tinggal tambah 1 method lagi, yaitu
->**withQueryString()**

```
"posts" => Post::latest()->filter(request(['search', 'category',
'author']))->paginate(7)->withQueryString()
```

* Jadi apapun yang ada di Query string sebelumnya, BAWA.. selesai...!!!

Dan sekarang kategori dan search-nya ikut kebawa

<http://127.0.0.1:8000/posts?category=web-programming&page=2>

<http://127.0.0.1:8000/posts?category=web-programming&search=Eveniet&page=2>

KEREN BGT !!!

14. View Login & Registration

Membuat View Login

Dokumentasi source-code example bootstrap :

<https://getbootstrap.com/docs/5.0/examples/>

Dokumentasi icons bootstrap :

<https://icons.getbootstrap.com/#install>

Kita tambahkan link CDN icons bootstrap di head
resource > views > layouts > main.blade.php

```
{-- Bootstrap Icons --}
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.3/font/bootstrap-icons.min.css">
```

Update file navbar kita

resources > views > partials > navbar.blade.php

```
<ul class="navbar-nav ms-auto">
    <li class="nav-item">
        <a href="/login" class="nav-link {{ $active === "login" ? 'active' : '' }}>
            <i class="bi bi-box-arrow-in-right"></i>
            Login
        </a>
    </li>
</ul>
```

Sekarang kita bikin dulu routes-nya login !!!

Masuk ke file **routes > web.php**

Tambahkan

```
Route::get('/login', [LoginController::class, 'index']);
```

Kita bikin controller-nya dulu !!!

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:controller LoginController
```

Atau kita pake plugin, bisa membuat model dengan perintah di **vscode**, dengan cara **ctrl + shift / command + P** untuk mengeluarkan command paletnya. kemudian ketikkan

```
>>> Artisan: Make Controller
```

Kemudian kita tulis nama controller-nya apa... misal : **LoginController** kemudian pilih tipe yang **Basic**

otomatis dibuatkan di dalam folder controller.

Setelah itu, jangan lupa import class-nya di file **routes > web.php** atau tambahkan namespace-nya

```
use App\Http\Controllers\LoginController;
```

* namespace yang tidak digunakan, bisa dihapus saja.

Sekarang kita masuk ke LoginController kita
http > Controllers > LoginController.php

Kita bikin sebuah **method** yang namanya **index()**

```
public function index()
```

```
{  
    return view('login.index', [  
        'title' => 'Login',  
        'active' => 'login'  
    ]);  
}
```

* view-nya kita kasih nama index, dan agar lebih rapi akan kita simpan ke dalam folder-folder terpisah nanti-nya.

folder login > file index.blade

sekarang kalau kita cek halaman login, error-nya akan berubah menjadi seperti ini

InvalidArgumentException
View [login.index] not found.

Karena view-nya belum ada, tapi routes-nya udah ada

Sekarang kita bikin view-nya !!!

Kita bikin folder dan file view-nya kita

resources > views > login > index.blade.php

```
@extends('layouts.main')  
  
@section('container')  
  
<div class="row justify-content-center">  
    <div class="col-md-4">  
        <main class="form-signin">  
  
            ... //copy dari example bootstrap - sign in  
  
        </form>  
        <small class="d-block text-center mt-3">Not registered? <a href="/register">Register Now!</a></small>  
    </main>  
    </div>  
</div>
```

```
@endsection
```

Kita copy CSS yang dari example sign-in bootstrap ke file CSS kita sendiri
public > css > style.css

```
.form-signin .form-floating:focus-within {  
z-index: 2;  
}  
  
.form-signin input[type="email"] {  
margin-bottom: -1px;  
border-bottom-right-radius: 0;  
border-bottom-left-radius: 0;  
}  
  
.form-signin input[type="password"] {  
margin-bottom: 10px;  
border-top-left-radius: 0;  
border-top-right-radius: 0;  
}
```

Kita tambahkan link CSS kita ke main/head
resource > views > layouts > main.blade.php

```
{-- My Style --}  
<link rel="stylesheet" href="/css/style.css">
```

Sekarang kita cek halaman login kita udah oke !!!

Membuat View Registrasi

Kita bikin controller-nya dulu !!!

Kita buat dengan terminal

```
$ php artisan make:controller RegisterController
```

atau bisa dengan command pallet

Sekarang kita masuk ke Register Controller kita

http > Controllers > RegisterController.php

Kita bikin sebuah **method** yang namanya **index()**

```
public function index()
{
    return view('register.index', [
        'title' => 'Register',
        'active' => 'register'
    );
}
```

Masuk lagi ke file **routes > web.php**

copy ke bawah dari yang login (Alt + Shift + Bawah)

```
Route::get('/register', [RegisterController::class, 'index']);
```

Setelah itu, jangan lupa import class-nya di file **routes > web.php**
atau tambahkan namespace-nya

```
use App\Http\Controllers\RegisterController;
```

Sekarang kita bikin view-nya !!!

Kita bikin folder dan file view-nya kita
resources > views > register > index.blade.php

* kita copy dari view-n ya login, dan kita edit
kita sesuaikan form input-nya dengan kebutuhan di database User

kita kasih **atribut nama** di semua input... **name="username"**, dll

kita tambahkan class rounded-top di input pertama dan rounded-bottom di input terakhir

Kita tambahkan style CSS ke file CSS kita sendiri
agar semua input, border tidak ada radius-nya
public > css > style.css

```
.form-registration input {  
    border-radius: 0;  
    margin-bottom: -1px;  
}
```

* Kalau setelah update CSS, dan halaman tidak berubah. Kita **refresh chace-nya (Ctrl + Shift + R)**

15. User Registration

Membuat Route & Method Register

Kita jalankan fungsi registrasi dulu sebelum login.

Kita bikin method di dalam registerController untuk mengelola data yang dikirimin.

Sebelum itu kita kasih action di form registrasi, yang mengarah ke halaman/route register. Tapi dengan method POST

```
<form action="/register" method="post">
```

Setelah itu, kita tambahkan di file **routes > web.php**

```
Route::post('/register', [RegisterController::class, 'store']);
```

* Kalau misalkan ada request ke halaman /register, tapi method-nya POST, maka panggil Controller register yang method-nya STORE

STORE biasanya digunakan ketika kita mau nyimpan, kalau nampilin biasanya kita tulis-nya CREAT, tapi itu hanya penamaan. bisa pakai index.

```
Route::get('/register', [RegisterController::class, 'create']);
Route::get('/register', [RegisterController::class, 'index']);
```

Kalau udah dikasih STORE, artinya kita akan punya sebuah method yang namanya STORE di Controller register.

kita tangkap dulu semua datanya yang dikirimin, terus kita tampilin dulu isinya apa.

caranya ada dua

kita bisa panggil **method request** untuk mengambil semua request yang ada dengan method **all()**

```
public function store()
{
    return request()->all();
}
```

Setelah itu, kalau kita kembali kembali ke web kita di halaman registrasi. Kita isikan data kita, saat kita klik tombol register, data-nya tidak tampil tapi muncul **ERROR 419 [PAGE EXPIRED]**

Cross-Site Request forgery (CSRF)

Error ini, sangat umum di Laravel. dan terjadi karena laravel-nya berusaha mengamankan halaman web kita, ketika kita melakukan POST lewat form.

Keamanan-nya seperti apa?

kita buka dokumentasi **CSRF** :

<https://laravel.com/docs/8.x/csrf#main-content>

Jadi [cross-site request forgery](#) (CSRF) adalah sebuah teknik serangan terhadap website kita, dimana biasanya serangan ini memalsukan request dari website lain (istilah-nya ngebajak request kita).

Jadi bisa aja, ada request yang dilakukan oleh website lain ke website kita menggunakan URL yang sesuai.

Jadi yang akan dikerjakan ke routes tertentu di dalam aplikasi kita, padahal itu request-nya bukan dari website kita, melainkan dari website orang lain.

Contoh-nya kayak gini :

```
<form action="https://your-application.com/user/email" method="POST">
    <input type="email" value="malicious-email@example.com">
</form>

<script>
```

```
document.forms[0].submit();  
</script>
```

* Jadi mungkin aja, ada website orang lain yang bikin form input, yang isinya (value) misalnya SCRIPT JAHAT, tapi URL-nya mengarah ke website kita. Dan hal tersebut akan dikerjain oleh controller register kita, padahal request-nya bukan dari website kita tapi **CROSS SITE** (dari website orang lain / NYEBRANG !!!)

Nah, laravel mencoba menangani itu.

Caranya <https://laravel.com/docs/8.x/csrf#preventing-csrf-requests>

Jadi kita akan menjaga request-nya agar selalu dikirimkan dari website kira saja, dengan menggunakan sesuatu yang disebut dengan **CRSF_TOKEN**.

Jadi website kita akan meng-generate sebuah token didalam session dan nanti akan dicocokkan dengan yang ada di request-nya.

CARANYA SIMPLE BANGET !!!

kita cuman butuh nambahin @csrf didalan FORM kita.

```
<form method="POST" action="/profile">  
    @csrf
```

Nah nanti di belakang layar, blade kita (@csrf adalah blade direktif) akan menerjemahkan seperti ini.

```
<!-- Equivalent to... -->  
<input type="hidden" name="_token" value="{{ csrf_token() }}" />  
</form>
```

Nanti biarkan laravel-nya yang akan nyocokin tokenya valid atau tidak.

Implementasi !!!

```
form action="/register" method="post">  
    @csrf
```

Dan **ERROR 419 [PAGE EXPIRED]** harusnya sudah hilang.
Dan kita sudah bisa menampilkan seluruh data beserta token-nya.

Form Validation

Sekarang data-nya sudah bisa dikirim semua datanya, tinggal kita lakukan validasi.

Karena kita butuh validasi, ketika data yang kita inputkan sudah benar atau belum.
Misalnya semuanya wajib di isi (required)

Sekarang, jika kita kosongi form-nya di halaman registrasi. Dan kita klik tombol register, maka data-nya masih bisa dikirim dengan nilai null.

Form Validation

Dokumentasi Validation :

<https://laravel.com/docs/8.x/validation#main-content>

Pertama, kita definisikan routes-nya
Kemudian, kita harus punya controller-nya.
di dalam controller kita harus punya method store(), dan kita validasi di dalam-nya.

Sebelumnya kita pakai cara ini

```
public function store()
{
    return request()->all();
}
```

Cara keduanya, kita juga bisa pakai variable request

```
public function store(Request $request)
{
    return $request->all();
}
```

* dan kedua cara tersebut sama saja. kalau melihat di dokumentasi menggunakan cara kedua.

Setelah itu, kalau kita kembali kembali ke web kita di halaman registrasi. Kita isikan data kita, kita klik tombol register dan data-nya muncul. Sama saja!

Isi method dari store() yaitu...

```
/**
 * Store a new blog post.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...
}
```

* jadi kita bikin agar si request-nya melalui method yang namanya validate(), dan didalamnya kita bikin sebuah array untuk ngasih tau rules atau aturan dari validasi kita.

Kalau wajib diisi, kita bisa kasih **required**

Berikan tanda pip (|) ketika kita mau nambahin rule-nya.

kalau title-nya harus unik (tidak boleh sama), maka tambahkan **unique:posts** (dari tabel apa harus unik-nya, biar nanti laravel-nya yang ngecek di tebelnya)

Maximal 255 (lebih dari itu ga boleh)

Rule apa saja yang bisa dituliskan di validasi?

Cek Dokumentasi-nya disini:

<https://laravel.com/docs/8.x/validation#available-validation-rules>

Kita coba implementasikan di aplikasi kita!

Sekarang kita masuk ke Register Controller kita

http > Controllers > RegisterController.php

```
public function store(Request $request)
{
    $request->validate([
        'name' => 'required|max:255',
        'username' => ['required', 'min:3', 'max:255', 'unique:users'],
        'email' => 'required|email|unique:users',
        'password' => 'required|min:5|max:255'
    ]);
}
```

* kita juga bisa mengganti tanda PIP-nya dengan array yang dipisah dengan titik koma (;)

* password ga perlu unik.. aneh juga kalo nanti “Password sudah ada yang pakai...” wkwk.

Nah, cerita-nya kalau misalnya dari validasi tersebut lolos, maka dia akan menjalankan apapun yang ada di bawahnya.

kita coba Dumb and Die,

```
dd('registrasi berhasil');
```

* tidak akan jalan ketika validasi diatas gagal !.

Setelah itu, kita coba Kita isikan data registrasi di web kita.

Jika data-nya sudah sesuai, maka **akan muncul “registrasi berhasil”**.

Dan tidak akan terjadi apa-apa jika tidak sesuai aturan.

Menangani Error Saat Validasi

Jadi sekarang validasi-nya sudah jalan, tapi problem-nya adalah kita ngga dikasih tau salah-nya dimana.

Kan keren kalau misalkan kita di kasih tau.

Sebetul-nya laravel udah ngasih kita pesan error-nya, cuman masih belum kita pake aja.

Nah pesan error-nya ini ada di directive blade yang namanya **@error**

kita bisa ngasih **class is-invalid** di input form-nya

```
<input class="is-invalid">
```

kita bikin supaya **class is-invalid** ini muncul hanya ketika ada error dengan memberi **@error**

```
<input class="@error('name') is-invalid @enderror">
```

Selain itu kita akan kasih tau error-nya apa...

Dokumentasi Bootstrap:

<https://getbootstrap.com/docs/5.0/forms/validation/#server-side>

Dokumentasi Laravel:

<https://laravel.com/docs/8.x/validation#quick-displaying-the-validation-errors>

```
@error('name')
    <div class="invalid-feedback">
        {{ $message }}
    </div>
@enderror
```

Kita terapkan di semua form input

Dan kita coba isikan data registrasi yang salah (tidak sesuai validasi) di web kita.

Dan akan muncul pesan error-nya !!! .

Nah, untuk required kita bisa jagain agar error-nya di handle browser, dengan menambahkan **atribut required** di semua form input-nya.

Kemudian, saat kita inputkan email dengan **tanpa @**, akan di tolak oleh browser karena type di input adalah email.

Tapi **akan lolos** meskipun tidak terdapat domain-nya semisal @gmail.com

Dokumentasi rule email validation :

<https://laravel.com/docs/8.x/validation#rule-email>

```
'email' => 'email:rfc,dns'
```

* agar domain yang dimasukkan benar, kita bisa menambahkan DNS

Agar inputan lama tidak hilang saat terjadi Error

Agar inputan lama tidak hilang saat ada error, bisa kita berikan value dengan method old didalamnya.

```
<input value="{{ old('username') }}">
```

Kita terapkan di semua form input **kecuali password**, karena sebaiknya password di isi lagi oleh user-nya.

Dan kita coba isikan data registrasi dengan benar kecuali 1 input, maka selain input form yang error masih tetap ada isi-nya dan tidak kosong lagi.

Insert data user ke database

Sekarang tinggal kita masukkan data-nya ke database.

Kita asumsikan data yang diinputkan sudah lolos validasi.

Kita simpan dulu ke dalam variabel

Kemudian kita kirim menggunakan **method create()** seperti di awal-awal menggunakan tinker

http > Controllers > RegisterController.php

```
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'username' => ['required', 'min:3', 'max:255', 'unique:users'],
        'email' => 'required|email:dns|unique:users',
        'password' => 'required|min:5|max:255'
    ]);

    // dd('registrasi berhasil');
    User::create($validatedData);
}
```

Jangan lupa kita panggil namespace model-nya jika belum ada.

```
use App\Models\User;
```

SQLSTATE[HY000]: General error: 1364 Field 'username' doesn't have a default value

Hal ini dikarenakan, pada model User kita

app > Models >User.php

terdapat proteksi bahwa yang boleh diisi secara massal (Mass Assignment), hanya nama, email, dan password. Sedangkan kita **sekarang kita punya username**

```
protected $fillable = [  
    'name', 'email', 'password',  
];
```

Kita bisa tambahkan username

```
protected $fillable = [  
    'name',  
    'username',  
    'email',  
    'password',  
];
```

Atau kita juga bisa ganti \$fillable menjadi \$guarded, agar kita tidak perlu edit lagi file ini ketika kita nambahin field baru.

```
protected $guarded = ['id'];
```

* Bisa juga di kosongin, tanpa id

Kemudian saat kita coba isikan lagi data registrasi yang benar, maka tidak akan terjadi apa-apa (blank page).

Tapi data-nya sudah masuk ke database.

Enkripsi Password

Namun, password yang kita kirim ke database belum kita enkripsi

Kalau bikin user yang ada password-nya, pastikan password-nya di enkripsi.

(Additional)

Menghapus 1 baris data di MySQL workbench

Klik kanan baris yang akan di hapus > **Delete Row (s) > Apply**

Kita tambahkan setelah validasi data kita.

http > Controllers > RegisterController.php

```
$validatedData['password'] = bcrypt($validatedData['password']);
```

Atau kita juga bisa pakai teknik hashing

Dokumentasi Hasing :

<https://laravel.com/docs/8.x/hashing#main-content>

```
$validatedData['password'] = Hash::make($validatedData['password']);
```

* Tambahkan namespace-nya kalau belum ada (error).

Agar setelah mensubmit form, tampilannya tidak kosong (blank) kita bikin sekalian redirect ke halaman login.

```
return redirect('/login');
```

Kemudian kita isikan lagi data registrasi yang benar, maka akan diarahkan ke halaman login dan data-nya sudah masuk ke database dengan **password yang terenkripsi**.

Flash Message

Dokumentasi Flashing :

<https://laravel.com/docs/8.x/session#flash-data>

Jadi kita bisa ngasih sebuah flash didalam session, agar pada saat diarahkan ke halaman login ada pesan “Registrasi berhasil, silakan login”

Caranya kita tinggal bikin request session flash, dengan nama flash-nya bebas (ex. status) kemudian pesan yang ditampilkan apa.

```
$request->session()->flash('status', 'Task was successful!');
```

Jadi , kita tambahkan **flash()**

```
$request->session()->flash('success', 'Registration successful! Please  
login');  
  
return redirect('/login');
```

Jadi begitu kita redirect, di dalam session sudah ada flash. tinggal kita tampilkan.

Kita kembali ke view halaman login
kita tampilkan pesan bentuknya alert yang bisa di close.

Dokumentasi Bootstrap Alert:

<https://getbootstrap.com/docs/5.0/components/alerts/#dismissing>

Kita kasih pengkondisian IF di dalam session dengan key yang nama-nya success

Kita bikin folder dan file view-nya kita
resources > views > login > index.blade.php

```
@if(session()->has('success'))  
  
    <div class="alert alert-success alert-dismissible fade show" role="alert">  
        {{ session('success') }}  
        <button type="button" class="btn-close" data-bs-dismiss="alert"  
aria-label="Close"></button>  
    </div>  
  
@endif
```

Kemudian saat kita isikan coba di web kita untuk registrasi, maka akan diarahkan ke halaman login dan **ditampilkan pesan berupa alert**.

Bahkan kita bisa perbaiki supaya lebih ringkas lagi.

Kita bisa langsung kirim sekalian redirect.

```
return redirect('/login')->with('success', 'Registration successful!  
Please login');
```

* redirect ke halaman login, dengan membawa flash message ini

16. User Login & Middleware

Sekarang saatnya mengaplikasikan fitur login-nya.

Konsep Authentication di dalam Laravel

Dokumentasi Authentication :

<https://laravel.com/docs/8.x/authentication#main-content>

Sebetulnya laravel sudah menyiapkan yang namanya **Starter kits**

Dokumentasi **Starter kits** :

<https://laravel.com/docs/8.x/authentication#starter-kits>

Jadi ada semacam plugin atau aplikasi yang khusus menangani masalah autentikasi, ada yang namanya **Laravel Breeze** dan juga **Laravel Jetstream**

Dokumentasi **Laravel Breeze** :

<https://laravel.com/docs/8.x/starter-kits#laravel-breeze>

Dokumentasi **Laravel Jetstream** :

<https://jetstream.laravel.com/introduction.html>

Jadi dengan menggunakan aplikasi ini, kita bisa dengan mudah menangani masalah autentikasi. Mulai dari registrasi user, login, bahkan fitur verifikasi lewat email, remember password, forgot password, dll

Jadi semua terkait autentikasi akan ditangani oleh starter kits ini.

Laravel Breeze bisa di install lewat composer, tapi UI-nya dibuat menggunakan Tailwind.

Pada tutorial ini, kita tidak menggunakan Starter Kits.

Kita akan pakai yang versi manual-nya saja.

<https://laravel.com/docs/8.x/authentication#authenticating-users>

Kalau kita tidak mau pakai starter kits, kita bisa menggunakan **Authentication Service** lewat **Facade**.

Facade ini seperti salah satu library didalam laravel yang nama-nya **Auth**.

Kita akan dipermudah dengan pengelolaan session untuk login-nya
Caranya cukup kita panggil namespace Facade di atas Controller kita dan akan kita pakai apa yang dibutuhin.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function authenticate(Request $request)
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our
records.',
        ]);
    }
}
```

- * kita butuh **method** yang nama-nya **authenticate()**
- * kita lakukan validasi terhadap form login.
- * Lalu kita jalankan sesuatu yang nama-nya **attemp()**, untuk ngecek apakah login yang dimasukan oleh user benar atau tidak.

Memperbaiki View Login

Kita buka lagi view login kita

resources > views > login > index.blade.php

```
<form action="/login" method="post">
@csrf
<div class="form-floating">
    <input type="email" name="email" class="form-control" id="email"
placeholder="name@example.com" autofocus required>
    <label for="email">Email address</label>
</div>
<div class="form-floating">
    <input type="password" name="password" class="form-control"
id="password" placeholder="Password" required>
    <label for="password">Password</label>
</div>

<button class="w-100 btn btn-lg btn-primary" type="submit">Login</button>
</form>
```

Membuat route & method Login

Kita buatkan routes nya

buka file routes > web.php kita tambahkan baris ini

```
Route::post('/login', [LoginController::class, 'authenticate']);
```

* nama method authenticate bebas saja, bisa login/store, disini mengikuti dokumentasi.

Sekarang kita masuk ke LoginController kita

http > Controllers > LoginController.php

Kita bikin sebuah **method** baru yang namanya **authenticate()**

```
use Illuminate\Support\Facades\Auth;

public function authenticate(Request $request)
{
    $request->validate([
        'email' => 'required|email:dns',
        'password' => 'required'
    ]);

    dd('berhasil login!');
}
```

Kemudian saat kita coba login di web kita, maka akan ditampilkan “berhasil login” jika yang diinputkan sudah benar, dan tidak akan terjadi apa-apa jika yang diinputkan masih belum sesuai validasi.

```
"berhasil login!"
```

Kita tambahkan pesan error di view login kita pada form email dan juga old value resources > views > login > index.blade.php

```
<input type="email" name="email" class="form-control @error('email')
is-invalid @enderror" id="email" placeholder="name@example.com" autofocus
required value="{{ old('email') }}">
<label for="email">Email address</label>

@error('email')
<div class="invalid-feedback">
    {{ $message }}
</div>
@enderror
```

Otentikasi Login

Setelah kita buat validasi-nya, kita bikin autentikasi-nya dengan menggunakan attempt().

Kalau login-nya berhasil, maka akan dipindahkan ke sebuah halaman, kalau gagal akan dikembalikan ke halaman login lagi dengan mengirimkan pesan error-nya.

Pesan error yang akan kita kirim tidak akan “email belum terdaftar” atau “password salah”. Walaupun sepertinya membantu user kalau password-nya salah. Tapi itu juga bisa menjadi celah keamanan user jahat jadi bisa tau kalau email-nya sudah terdaftar.

Sebaiknya kita jangan sedikitpun memberi clue !!!

Jadi kita cuma ngasih tau “Login gagal !”

```
if ($Auth::attempt($credentials)) {  
    $request->session()->regenerate();  
    return redirect()->intended('/dashboard');  
}
```

* Kita melakukan re-generate pada session untuk menghindari teknik hacking (kejahatan) session fixation

you should regenerate the user's session to prevent session fixation:

https://en.wikipedia.org/wiki/Session_fixation

Teknik ini merupakan cara bagaimana seorang hacker masuk ke dalam celah keamanan sistem menggunakan session (pura-pura masuk dengan session yang sama sebelumnya).

* Setelah itu kita melakukan redirect menggunakan intended() supaya melewati middleware

Kemudian, jika email/password-nya salah kita balikin ke halaman login sambil kita kirimkan error-nya.

Kalau kita menggunakan `withErrors()`, maka akan masuk ke variabel `@error('email')`

Tapi kalau kita mau masukin ke dalam flash message, kita cukup pakai `with()` saja.

```
return back()->with('loginError', 'Login failed!');
```

Sehingga di view login-nya kita tinggal **nambahin alert** lagi.

```
@if(session()->has('loginError'))  
  
    <div class="alert alert-danger alert-dismissible fade show" role="alert">  
        {{ session('loginError') }}  
        <button type="button" class="btn-close" data-bs-dismiss="alert"  
aria-label="Close"></button>  
    </div>  
  
@endif
```

Kemudian kita coba login di web kita dan akan muncul pesan “login failed” jika salah atau diarahkan ke halaman dashboard jika benar.

Membuat DashboardController

Kita bisa pakai **command pallet** di vscode, dengan cara **ctrl + shift / command + P** kemudian ketikkan

```
>>> artisan: make controller  
>>> DashboardController  
>>> basic
```

Atau bisa pakai terminal

```
$ php artisan make:controller DashboardController
```

* DashboardController = nama controller-nya

Kita tambahkan **method index()** untuk menampilkan view-nya.

```
public function index()
{
    return view('dashboard.index');
}
```

Kita buat folder dan file untuk index dashboard-nya
resources > views > dashboard > index.blade.php

```
<h1>Welcome, Conan Edogawa</h1>
```

Kemudian kita bikin route-nya.

routes > web.php

```
public function index()
{
    return view('dashboard.index');
}
```

Middleware

Sekarang kita sudah bisa login, namun masih bisa ke halaman login. Dan kita masih belum bisa logout.

Dan tombol login yang di navbar juga akan kita hilangkan

Untuk melakukan itu, kita harus faham dulu mengenai sesuatu yang disebut middleware authentication.

Dokumentasi **Middleware** :

<https://laravel.com/docs/8.x/middleware#main-content>

Middleware provide a convenient mechanism for inspecting and filtering HTTP requests entering your application. For example, Laravel includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to your application's login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Middleware menyediakan mekanisme yang mudah untuk memeriksa dan memfilter permintaan HTTP yang masuk ke aplikasi Anda. Misalnya, Laravel menyertakan middleware yang memverifikasi bahwa pengguna aplikasi Anda telah diautentikasi. Jika pengguna tidak diautentikasi, middleware akan mengarahkan pengguna ke layar login aplikasi Anda. Namun, jika pengguna diautentikasi, middleware akan mengizinkan permintaan untuk melanjutkan lebih jauh ke dalam aplikasi.

Middleware bisa dipasang misalnya di routes kita.

* **Middleware** : sebuah software yang ada di tengah-tengah sesuatu.

Secara default, di dalam laravel udah banyak middleware yang otomatis jalan.

Dokumentasi **Global Middleware** :

<https://laravel.com/docs/8.x/middleware#global-middleware>

If you want a middleware to run during every HTTP request to your application, list the middleware class in the `$middleware` property of your `app/Http/Kernel.php` class.

Jika kita buka file `app/Http/Kernel.php` akan terdapat middleware yang sudah dijalankan otomatis.

* **auth** bisa kita panggil ketika halaman tertentu atau routing tertentu hanya bisa diakses oleh user yang sudah ter-autentikasi

* **guest** yang dijalankan ketika user-nya belum ter-autentikasi

Dan masih banyak lagi!

Menerapkan Middleware Auth

```
Route::get('/login', [LoginController::class,
'index'])->middleware('guest');
```

* Halaman Login hanya bisa diakses oleh user yang belum ter-autentikasi

Nah, jika kita sudah terautentikasi dan mencoba untuk kembali login, maka akan diarahkan ke halaman /home meskipun halamannya tidak ada (padahal yang seharusnya tidak bisa)

Hal itu terjadi karena default-nya ke home.

Untuk mengubahnya kita bisa ke file
app > Providers > RouteServiceProvider.php

```
// public const HOME = '/home';

public const HOME = '/';
```

* Jadi halaman home kita adalah rootnya (/)

Sekarang jika kita ulangi login lagi, maka akan diarahkan ke halaman / root

Mengganti tombol login saat sudah terautentikasi

Kita buka lagi navbar kita, **views > partials > navbar.blade.php**

Kita bisa gunakan sebuah directive yang namanya **@auth @endauth** kalau kita mau ngecek apakah seorang user sudah login atau belum. Atau **@guest @endguest** jika sebaliknya.

Atau bisa juga di gabung, **@auth @else @endauth**

Dokumentasi Navbar Dropdown :

<https://getbootstrap.com/docs/5.0/components/navbar/>

Icon untuk Logout :

<https://icons.getbootstrap.com/icons/box-arrow-right/>

```
<ul class="navbar-nav ms-auto">

    @auth

        <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">
                Welcome back, {{ auth() -> user() -> name }}
            </a>
            <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
                <li><a class="dropdown-item" href="#"><i class="bi bi-layout-text-sidebar-reverse"></i> My Dashboard</a></li>
                <li><hr class="dropdown-divider"></li>
                <li>
                    <a class="dropdown-item" href="#"><i class="bi bi-box-arrow-right"></i> Logout</a>
                </li>
            </ul>
        </li>

    @else

        <li class="nav-item">
            <a href="/login" class="nav-link {{ ($active === "login") ? 'active' : '' }}">
                <i class="bi bi-box-arrow-in-right"></i>
                Login
            </a>
        </li>

    @endauth

</ul>
```

* Sekarang tombol login sudah di ganti dengan welcom, nama user dan dropdown untuk ke dashboard dan logout

Membuat Logout

Untuk logout, kita **butuh form**.

Karena jalannya **harus ada CSRF** lagi.

Kita butuh form yang isinya hanya button dengan **type submit**, tapi kita buat supaya tidak seperti button.

```
<form action="/logout" method="post">
    @csrf
    <button type="submit" class="dropdown-item"><i class="bi bi-box-arrow-right"></i> Logout</button>
</form>
```

* jangan lupa, setiap form butuh @csrf

Kemudian kita bikin route untuk logout-nya dulu.

buka file **routes > web.php** kita tambahkan baris ini

```
Route::post('/logout', [LoginController::class, 'logout']);
```

Dokumentasi Logout :

<https://laravel.com/docs/8.x/authentication#logging-out>

Sekarang kita bikin method-nya di LoginController

http > Controllers > LoginController.php

```
public function logout(Request $request)
{
    Auth::logout();
    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect('/');
}
```

- * Pastikan **namespace Auth** sudah ada
- * invalidate session-nya supaya ngga bisa di pakai.
- * bikin baru token session-nya supaya ngga di bajak.
- * kemudian kita belikin mau ke halaman mana, semisal home.

Kalau tidak mau pakai **Request \$request** bisa pakai ini

```
public function logout()
{
    Auth::logout();

    request()->session()->invalidate();

    request()->session()->regenerateToken();

    return redirect('/');
}
```

Kemudian saat kita coba logout, di halaman home otomatis berubah menjadi tombol login lagi. dan kita bisa masuk ke halaman login.

Memperbaiki akses halaman dashboard & Named Route

Nah, meskipun setelah logout kita masih bisa masuk ke halaman dashboard. Karena belum kita kasih tau di routes-nya bahwa halaman dashboard hanya boleh diakses oleh orang yang sudah login.

Jadi kita tambahkan di routes kita **routes > web.php**

```
Route::get('/login', [LoginController::class,
'index'])->middleware('guest');

Route::post('/login', [LoginController::class, 'authenticate']);
Route::post('/logout', [LoginController::class, 'logout']);
```

```
Route::get('/register', [RegisterController::class,
'index'])->middleware('guest');

Route::post('/register', [RegisterController::class, 'store']);

Route::get('/dashboard', [DashboardController::class,
'index'])->middleware('auth');
```

Namun sekarang jika kita coba masuk ke halaman dashboard, maka akan muncul error

Symfony\Component\Routing\Exception\RouteNotFoundException
Route [login] not defined.

Hal ini dikarenakan kita ngga punya route yang namanya **[login]**

Kalau kita masuk ke **app > Http > Middleware > Authenticate.php**

```
protected function redirectTo($request)
{
    if (! $request->expectsJson()) {
        return route('login');
    }
}
```

Defaultnya, bahwa kalau misalnya ada user yang ngga terautentikasi berusaha masuk ke halaman yang di autentikasi, maka redirect dia ke route yang namanya login.

Named Route

Dokumentasi **Named Routes** :

<https://laravel.com/docs/8.x/routing#named-routes>

Route itu bisa kita kasih nama, atau istilah di laravel itu **Named Route**

Supaya tidak berpatokan pada URL-nya

Karena user yang belum terautentikasi redirect ke route yang namanya login. Jadi harus kita kasih dahulu dimana route login-nya

```
Route::get('/login', [LoginController::class,
'index'])->name('login')->middleware('guest');
```

Nah sekarang jika ada yang iseng coba masuk ke halaman /dashboard, maka akan diarahkan ke halaman login.

17. Dashboard UI

Download Template Bootstrap

Dokumentasi Example Dashboard Bootstrap :

<https://getbootstrap.com/docs/5.0/examples/dashboard/>

Download Examples (sebelumnya sudah, untuk login)

Kita copy-kan file (bootstrap) **dashboard > dashboard.css** ke folder **public > css**

Kita copy-kan file (bootstrap) **dashboard > dashboard.js** ke folder **public > js**

Menghapus controller dashboard

Sebelumnya, kita sudah membuat routes di **web.php** untuk menyimpan halaman dashboard sebagai controller.

Pak dhika memutuskan untuk mengelola data postingan nantinya tidak di controller dashboard, tapi hanya akan digunakan untuk **menampilkan** halaman dashboard-nya saja.

Untuk pengelolaan post-nya akan di simpan di controller terpisah.

method index pada DashboardController akan dipindahkan agar dikelola oleh route dengan menggunakan closure saja.

```
Route::get('/dashboard', function() {
    return view('dashboard.index');
})->middleware('auth');
```

dan file **DashboardController.php** bisa dihapus saja.

Membuat & Mengubah view dashboard

Kita copy-kan isi code dari (bootstrap) **dashboard > dashboard.html** ke file **resources > views > dashboard > index.blade.php**

Mengubah view menjadi layout

kita pisah-pisah file-nya

Menggunakan Feather Icons

Dokumentasi :

<https://feathericons.com/>

Membuat halaman My Posts (Resource Controller)

kita arahkan My Posts ke **route** /dashboard/posts yang akan mengarah ke controller baru, dan controller tersebut akan kita pakai untuk mengelola semua data post oleh user tertentu.

User bisa menambah post, edit post, melihat detail post, dan juga delete (CRUD).

Untuk melakukan hal tersebut, kita akan bikin sebuah routes yang akan mengarah ke controller yang kita sebut dengan resource controller.

Apa itu resource controller?

Yaitu sebuah controller yang sudah otomatis mengelola data CRUD, sehingga kita tidak perlu lagi manual mencari tahu route-nya apa.

Jadi kalau mau ke tambah post, kita punya /dashboard/posts/create (misalnya). Nah kita ngga perlu bikin manual, udah otomatis di bikinin sama resource-nya.

Dokumentasi resource controller :

<https://laravel.com/docs/8.x/controllers#resource-controllers>

If you think of each Eloquent model in your application as a "resource", it is typical to perform the same sets of actions against each resource in your application. For example, imagine your application contains a Photo model and a Movie model. It is likely that users can create, read, update, or delete these resources.

Because of this common use case, Laravel resource routing assigns the typical create, read, update, and delete ("CRUD") routes to a controller with a single line of code. To get started, we can use the make:controller Artisan command's --resource option to quickly create a controller to handle these actions:

Jika Anda menganggap setiap model Eloquent dalam aplikasi Anda sebagai "sumber daya", biasanya Anda akan melakukan serangkaian tindakan yang sama terhadap setiap sumber daya dalam aplikasi Anda. Misalnya, bayangkan aplikasi Anda berisi model Foto dan model Film. Kemungkinan besar pengguna dapat membuat, membaca, memperbarui, atau menghapus sumber daya ini.

Karena kasus penggunaan umum ini, perutean sumber daya Laravel menetapkan rute pembuatan, pembacaan, pembaruan, dan penghapusan ("CRUD") yang umum ke pengontrol dengan satu baris kode. Untuk memulai, kita dapat menggunakan opsi --resource dari perintah make:controller Artisan untuk membuat pengontrol dengan cepat guna menangani tindakan ini:

```
$ php artisan make:controller PhotoController --resource
```

Sehingga nanti akan otomatis dibuatkan controller yang sudah punya route ke path berikut :

Actions Handled By Resource Controller

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

Nah, nanti di dalam route **web.php**-nya kita tidak perlu bikin satu-satu untuk tiap route di atas.

Cukup 1 baris menangani semua.

```
use App\Http\Controllers\PhotoController;  
  
Route::resource('photos', PhotoController::class);
```

Specifying The Resource Model

If you are using [route model binding](#) and would like the resource controller's methods to type-hint a model instance, you may use the `--model` option when generating the controller:

Jadi kita bisa langsung menentukan route model binding untuk controller baru kita. Kita akan ngasih tau bahwa controller yang kita punya, selain bentuknya resource dia juga langsung terhubung ke model (yang kita inginkan), dalam hal ini model kita adalah Post

```
$ php artisan make:controller PhotoController --model=Photo --resource
```

Membuat resource controller

Kita Implementasikan bikin resource controller-nya dulu !!!

Kita akan namakan **DashboardPostController** supaya tidak tertukar dengan **PostController** sebelumnya

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:controller DashboardPostController --model=Post  
--resource
```

Atau kita pake plugin, bisa membuat model dengan perintah di **vscode**, dengan cara **ctrl + shift / command + P** untuk mengeluarkan command paletnya. kemudian ketikkan

```
>>> Artisan: Make Controller
```

Kemudian kita tulis nama controller-nya apa... misal : **DashboardPostController** kemudian pilih tipe yang **Resource**

Apakah mau me-reference ke Model? pilih **YES**, kemudian kita tulis model-nya apa misal : **Post**

otomatis akan dibuatkan controller baru di dalam folder controller yang sudah terdapat method untuk CRUD dan sudah terkoneksi dengan model Post.

```
public function index()
{
    // untuk menampilkan semua data post berdasarkan user tertentu
}

public function create()
{
    // untuk menampilkan halaman tambah postingan
}

public function store(Request $request)
{
    // untuk menjalankan fungsi tambah-nya
}

public function show(Post $post)
{
    // fungsi lihat detail dari sebuah postingan
    // dan sudah otomatis ada route model binding-nya (Post $post)
}

public function edit(Post $post)
{
    // halaman buat nampilin ubah data
}

public function update(Request $request, Post $post)
{
    // halaman untuk proses ubah data-nya
}

public function destroy(Post $post)
{
    // untuk delete postingannya
}
```

Membuat route untuk resource controller

Pada dokumentasi, contohnya seperti ini

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Jadi kita tambahkan di routes kita **routes > web.php**

```
Route::resource('/dashboard/posts',
DashboardPostController::class)->middleware('auth');
```

Kemudian kita bisa memastikan apakah halaman /dashboard/posts sudah di tangani oleh DashboardPostController seperti ini dahulu

```
public function index()
{
    return 'ini halaman dashboard post';
}
```

Sekarang kita arahkan ke view

```
public function index()
{
    return view('dashboard.posts.index');
}
```

Kemudian kita buat folder dan file baru untuk index dashboard post
resources > views > dashboard > posts > index.blade.php

yang isi-nya kita samakan dulu dengan index dashboard-nya.

Setelah itu kita cek di web kita

<http://127.0.0.1:8000/dashboard/posts>

Memperbaiki menu yang active

Kita kembali ke **halaman sidebar**, kita akan kasih active-nya dinamis sesuai dengan halaman yang lagi aktif.

resources > views > dashboard > layouts > sidebar.blade.php

Pada video sebelum-nya ada yang ngasih saran di kolom komentar.

Cara active-nya ngga perlu bikin variabel baru, tapi kita bisa cek lewat REQUEST

Caranya yaitu

```
 {{ Request::is('dashboard') ? 'active' : '' }}
```

Full-nya halaman sidebar

```
<nav id="sidebarMenu" class="col-md-3 col-lg-2 d-md-block bg-light sidebar collapse">
    <div class="position-sticky pt-3">

        <ul class="nav flex-column">
            <li class="nav-item">
                <a class="nav-link {{ Request::is('dashboard') ? 'active' : '' }}" aria-current="page" href="/dashboard">
                    <span data-feather="home"></span>
                    Dashboard
                </a>
            </li>
            <li class="nav-item">
                <a class="nav-link {{ Request::is('dashboard/posts') ? 'active' : '' }}" href="/dashboard/posts">
                    <span data-feather="file-text"></span>
                    My Posts
                </a>
            </li>
        </ul>

    </div>
```

```
</nav>
```

Kita bisa ganti navbar active yang sebelumnya dengan cara ini, **karena lebih CLEAN.**

Menampilkan data post berdasarkan user yang login

kita pakai tabel yang dari template dashboard sebelumnya.

Akan kita looping data dari tabel

Caranya untuk mendapatkan data dari tabelnya, kita harus ngirimin dulu di method index pada **DashboardPostController**

Semisal kita coba menampilkan semua post yang ada di dalam database.

```
public function index()
{
    return Post::all();
    return view('dashboard.posts.index', [
        'post' => Post::all()
    ]);
}
```

Untuk menampilkan postingan sesuai dengan user-nya, kita harus QUERY

```
public function index()
{
    return Post::where('user_id', auth()->user()->id)->get();
}
```

pastikan user sudah memiliki postingan.
jika tidak, maka akan tampil []

Email invalid error

Saat kita coba login menggunakan email dengan **domain @example.org**, maka email akan dianggap invalid/domain tersebut tidak ada oleh laravel karena kita validasi email dengan DNS

untuk itu kita edit di database emailnya menjadi **@gmail.com**

Jika sudah terdapat postingan pada user, sekarang kita kirim data-nya

```
public function index()
{
    return view('dashboard.posts.index', [
        'posts' => Post::where('user_id', auth()->user()->id)->get()
    ]);
}
```

Lalu kita looping <tr> nya pada tabel di index dashboard posts-nya
resources > views > dashboard > posts > index.blade.php

```
@foreach ($posts as $post)
<tr>
    <td>1,001</td>
    <td>random</td>
    <td>data</td>
    <td>placeholder</td>
    <td>text</td>
</tr>
@endforeach
```

* Nah, jika begini maka data pada , <td> tersebut akan di ulang sebanyak data/postingan yang dimiliki user yang login.

Tinggal kita isi dalemnya menggunakan data dari post-nya

```
@foreach ($posts as $post)

    <tr>
        <td>{{ $loop->iteration }}</td>
        <td>{{ $post->title }}</td>
        <td>{{ $post->category->name }}</td>
        <td>placeholder</td>
    </tr>

@endforeach
```

* Untuk menampilkan angka, kita bisa menggunakan **variabel \$loop** di blade directive-nya.

Dokumentasi Loops :

<https://laravel.com/docs/8.x/blade#loops>

untuk mengambil angka looping yang dimulai dari 1, maka kita bisa pakai

```
 {{ $loop->iteration }}
 //untuk mengambil angka looping yang dimulai dari 1

 {{ $loop->index }}
 //untuk mengambil angka looping yang dimulai dari 0

 {{ $loop->remaining }}
 //untuk mengambil angka sisa looping/angka dari tertinggi ke 0
```

* Selengkapnya ada pada dokumentasi.

Setelah itu kita membuat tombol aksi

```
<td>
  <a href="/dashboard/posts/{{ $post->id }}" class="badge bg-info"><span
data-feather="eye"></span></a>
  <a href="" class="badge bg-warning"><span data-feather="edit"></span></a>
  <a href="" class="badge bg-danger"><span data-feather="x-circle"></span></a>
</td>
```

Membuat halaman detail post berdasarkan user yang login

Untuk membuat halaman detail, kita sudah punya URL-nya

```
/dashboard/posts/{{ $post->id }}
```

Jadi tidak perlu kita tangani didalam routes-nya, karena sudah ditangani oleh route resource. Tinggal kita tangani di bagian **method show()** pada **DashboardPostController**

Sekarang kita coba return \$post;

```
public function show(Post $post)
{
    return $post;
}
```

dengan begitu, kita bisa mendapatkan data detail postingan namun dengan id.

<http://127.0.0.1:8000/dashboard/posts/1>

Bagaimana supaya kita bisa pakai slug?

Kalau pakai resource, kita tidak bisa masukin route model binding seperti ini semisal gini (jadi tidak jalan)

```
// tidak bisa masukin route model binding seperti ini {post:slug}  
untuk resource  
Route::resource('/dashboard/posts/{post:slug}',  
DashboardPostController::class)->middleware('auth');
```

* karena URL-nya bukan itu.

Sebenarnya bisa bikin route baru (kita timpa istilahnya), tapi sayang-nya kita udah pake resource

```
Route::get('/dashboard/posts/{post:slug}')
```

Supaya kita bisa pakai slug, **ada cara ngakalinnya!!**
yaitu dengan membuat slug itu sebagai nilai default untuk pencarian.

Jadi meskipun ngga pakai route model binding seperti di atas, yang dicari tuh otomatis slug.
biasanya yang dicari itu id.

Dokumentasi Route model binding :

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

Dokumentasi Custom key name :

<https://laravel.com/docs/8.x/routing#customizing-the-default-key-name>

kadang-kadang, kita pengennya agar si eloquent ini mencari model menggunakan kolom selain id.

Untuk melakukan ini, kita harus kasih tau kolomnya di dalam model.
Nah caranya kita bisa tulis di route parameter

```
use App\Models\Post;
```

```
Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

* ini yang sudah kita coba sebelumnya.

Kalau kita pengennya si modelnya itu selalu menggunakan kolom di database selain id (tidak manual), caranya kita akan timpa menggunakan sebuah method **getRouteKeyName** di dalam model kita.

Jadi nanti di dalam model-nya ada method ini

```
/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

Implementasi

Kita copy method tersebut, kemudian kita tambahkan di Model Post kita.

app > Models > Post.php

```
public function getRouteKeyName()
{
    return 'slug';
}
```

* Nah sekarang setiap route, otomatis **mencari-nya slug, bukan lagi id.**

Namun, sekarang kita tidak lagi bisa mencari postingan berdasarkan id-nya.

Kemudian, kita akan tampilkan datanya dengan view

```
public function show(Post $post)
{
    return view('dashboard.posts.show', [
        'post' => $post
    ]);
}
```

Kita buat view-nya

resources > views > dashboard > posts > show.blade.php

[Finish]

18. Create Post Form

Membuat tombol create post

kita arahkan ke route /create agar nanti bisa diambil oleh route resource kita untuk menampilkan form tambah data post-nya. (sudah default)

```
<a href="/dashboard/posts/create" class="btn btn-primary  
mb-3">Create New Post</a>
```

Dan jika kita masuk ke halaman /create, tidak 404 not found karena sudah ditangani oleh resource. Namun masih kosongan

Sekarang kita arahkan pada sebuah view di method create() pada DashboardPostController kita

```
public function create()  
{  
    return view('dashboard.posts.create');  
}
```

Membuat view create

Kita buat view-nya

resources > views > dashboard > posts > create.blade.php

```
@extends('dashboard.layouts.main')  
  
@section('container')
```

```
<div class="d-flex justify-content-between flex-wrap flex-md-nnowrap align-items-center pt-3 pb-2 mb-3 border-bottom">
    <h1 class="h2">Create New Post</h1>
</div>

@endsection
```

Mengaktifkan link my posts pada sidebar untuk sub halaman

Namun pada halaman /create, link my posts pada sidebar tidak aktif.

Untuk mengaktifkan link-nya, kita ubah pada sidebar menjadi seperti ini.

```
 {{ Request::is('dashboard/posts*') ? 'active' : '' }}
```

* kita tambahkan wildcard sebuah bintang (*), nanti dia akan melihat apapun yang ada setelah tanda post, akan membuat halaman tersebut aktif.

Karena halaman create dan update adalah sub dari halaman posts. maka dari itu kita buat link my posts pada sidebar menjadi aktif.

Membuat form

Kita copy form-nya dari bootstrap :

<https://getbootstrap.com/docs/5.0/forms/overview/>

```
<div class="col-lg-8">

    <form>
        <div class="mb-3">
            <label for="exampleInputEmail1" class="form-label">Email
            address</label>
            <input type="email" class="form-control" id="exampleInputEmail1"
            aria-describedby="emailHelp">
        </div>
```

```
<button type="submit" class="btn btn-primary">Submit</button>
</form>

</div>
```

Form-nya method-nya pasi POST, action-nya mengarah ke route yang akan memproses data-nya.

```
<form method="POST" action="/dashboard/posts">
```

dengan begitu, akan otomatis mengarah ke method **store()** pada DashboardPostController kita kalau kita pakai resource. (jadi enak banget, ngga perlu bikin lagi route di web.php -nya.. semua udah di tangani)

```
Route::resource('/dashboard/posts',
DashboardPostController::class)->middleware('auth');
```

- * Begitu halamannya ke /dashboard/posts dan method-nya GET, maka dia langsung ke **index()**
- * Begitu halamannya ke /dashboard/posts dan method-nya POST, maka dia langsung ke **store()**
- * Begitu halamannya ke /dashboard/posts dan method-nya PUT, maka dia langsung ke **update()**
- * Begitu halamannya ke /dashboard/posts dan method-nya DELETE, maka dia langsung ke **destroy()**

Kita sesuaikan form-nya.

Membuat slug otomatis menggunakan Sluggable

Jadi saat kita mengisikan sebuah judul, dan kemudian pindah ke form slug. Maka akan otomatis terisikan slug-nya

Semisal judul-nya Web Programming UNPAS, maka akan dibuatkan **web-programming-unpas**

Gimana caranya?

Kita butuh bantuan sebuah package didalam composer (nanti kita tarik/download) yang tugas-nya memang untuk membuat slug.

Eloquent Sluggable

Eloquent Sluggable (by: cviebrock) :

<https://github.com/cviebrock/eloquent-sluggable>

Instalasi :

```
$ composer require cviebrock/eloquent-sluggable
```

* tambahkan jika tidak sesuai requirement
--ignore-platform-reqs

Akan ada warning jika terdapat Model yang tidak sesuai. Seperti semisal Post_.php (terdapat underscore)

Cara menggunakan Sluggable

Yang pertama, kita harus update dulu Model kita (cari model mana yang mau pakai slug-nya)

Pada case kita, model yang mau kita pakai yaitu Model Post.

app > Models > Post.php

Updating your Eloquent Models

Your models should use the Sluggable trait, which has an abstract method sluggable() that you need to define. This is where any model-specific configuration is set (see [Configuration](#) below for details):

```
use Cviebrock\EloquentSluggable\Sluggable;

class Post extends Model
{
    use Sluggable;

    /**
     * Return the sluggable configuration array for this model.
     *
     * @return array
     */
    public function sluggable(): array
    {
        return [
            'slug' => [
                'source' => 'title'
            ]
        ];
    }
}
```

* Kita include-kan namespace-nya

```
use Cviebrock\EloquentSluggable\Sluggable;
```

* terus kita panggil trait-nya (supaya bisa dipakai)

```
use Sluggable;
```

* Jika terdapat 2 trait atau lebih, bisa kita pisahkan dengan koma (,)

```
use HasFactory, Sluggable;
```

* trait itu wajib kita bikin method-nya (wajib digunakan)!

* Lalu kita harus bikin method yang nama-nya sluggable() dan tentukan source-nya mau ngambil dari field apa? (di dalam tabel Post kita)

```
public function sluggable(): array
{
    return [
        'slug' => [
            'source' => 'title'
        ]
    ];
}
```

Pada case kita, kita ngambil dari field “title” di database kita.

Dan jika sudah diterapkan pada Model Post kita, maka sudah bisa dipakai

Sekarang cara pakai-nya gimana?

Cara pakain ya, karena kita pengennya pada saat mengetikkan judul.. Lalu pada saat inputan ke slug akan otomatis dibuatkan slug-nya, jadi mau tidak mau kita mesti pakai Javascript.

Kita butuh melakukan ini dengan AJAX atau pakai fetch API, supaya otomatis begitu kita tab (pindah inputan) dia manggil method sluggable(). Otomatis ambil title-nya terus ubah jadi slug.

Berarti kita butuh Javascript, kita bisa balik ke halaman view create (**resources > views > dashboard > posts > create.blade.php**)

tambahkan <script> sebelum @endsection

Kita harus tau caranya menggunakan fetch API

Pak sandhika biasanya baca tutorial di

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Atau bisa mengikuti playlist Javascript Lanjutan

Mau pakai XML HTTP request versi AJAX juga boleh, tapi pak sandhika pengennya simple. jadi akan pakai fetch saja.

Kita bikin 2 buah const dulu

```
<script>

const title = document.querySelector('#title');
const slug = document.querySelector('#slug');

</script>
```

Kemudian kita bikin sebuah eventHandler yang menangani ketika yang kita tuliskan di dalam title itu berubah, berarti kita butuh event-nya onChange.

Caranya gini!

```
<script>

const title = document.querySelector('#title');
const slug = document.querySelector('#slug');

title.addEventListener('change', function() {

    //callback
    fetch('/dashboard/posts/checkSlug?title=' + title.value)

        .then (response => response.json())
        //datanya masih promise, jadi kita then lagi

        .then (data => slug.value = data.slug)
    })

</script>
```

Sebenarnya kita bisa pakai OnKeyUp, tapi itu pemanggilan fetch-nya nanti ketika kita mengetik sesuatu. Jadi 1 tombol langsung request.

Tapi kalau kita mau nunggu selesai dulu, terus kita pindah tab. Kita pakainya change. Supaya ngga terlalu sering pemanggilan-nya

Kita buat method baru yang tugas-nya untuk menangani ketika ada permintaan slug.

```
use App\Models\Post;
use Illuminate\Http\Request;
use \Cviebrock\EloquentSluggable\Services\SlugService;

public function checkSlug(Request $request)
{
    $slug = SlugService::createSlug(Post::class, 'slug',
$request->title);
    return response()->json(['slug' => $slug]);
}
```

* jangan lupa tambahkan namespace-nya

Dokumentasi slug service :

<https://github.com/cviebrock/eloquent-sluggable?tab=readme-ov-file#the-slugservice-class>

```
use \Cviebrock\EloquentSluggable\Services\SlugService;

$slug = SlugService::createSlug(Post::class, 'slug', 'My First Post');
```

* Dengan begitu, bahkan kerennya dia akan ngecek di database apakah ada slug yang sama, karena slug itu harus unique. Dan hal tersebut sudah dikerjakan oleh library sluggable-nya.

Kita perlu bikin route baru (manual) karena bukan bagian dari resource.

```
Route::get('/dashboard/posts/checkSlug',
[DashboardPostController::class, 'checkSlug'])->middleware('auth');
```

* Pastikan route untuk checkSlug berada di atas route resource. (karena jika di bawahnya tidak akan jalan)

```
Route::get('/dashboard/posts/checkSlug',
[DashboardPostController::class, 'checkSlug'])->middleware('auth');

Route::resource('/dashboard/posts',
DashboardPostController::class)->middleware('auth');
```

Kita bisa bikin form input untuk slug-nya read only/disable (tidak bisa diganti) Tapi kalau semisal boleh meng-custom juga bisa.

Tinggal menambahkan < disable readonly >

Dokumentasi disable form control :

<https://getbootstrap.com/docs/5.0/forms/form-control/#disabled>

```
<input type="text" class="form-control" id="slug" name="slug"
disabled readonly>
```

Membuat combo box Category

Bootstrap Form Select :

<https://getbootstrap.com/docs/5.0/forms/select/>

```
<div class="mb-3">
    <label for="category" class="form-label">Slug</label>
    <select class="form-select" name="category_id">

        @foreach ($categories as $category)

            <option value="{{ $category->id }}>{{ $category->name }}</option>

        @endforeach

    </select>
</div>
```

* Kalau kita pengen bisa looping dari tabel category, itu caranya kita bisa include-kan category-nya di controller dulu. Nanti kita kirimkan ke method creat-nya.

Jadi kita panggil dulu Model-nya
DashboardPostController

```
use App\Models\Category;
```

kemudian kita kirimin ke method create()

```
return view('dashboard.posts.create', [
    'categories' => Category::all()
]);
```

* sekarang sudah muncul data kategori-nya

Membuat editor untuk body menggunakan Trix Editor

Sekarang, kita butuh sebuah editor untuk nulisin body-nya.

Excerpt kita lewati, dan akan kita bikin supaya dia langsung ngambil beberapa kata dari body.

Trix Editor :

<https://trix-editor.org/>

by basecamp :

<https://github.com/basecamp/trix>

Sebenarnya masih banyak editor yang lain.

Kita juga bisa pakai ckeditor :

<https://ckeditor.com/>

Trix cukup simple

walaupun fitur untuk menambah file masih belum kita pakai.

Dengan Trix editor ini, kita bisa nulis blog-nya ada formatting-nya. Ada bold, italic, strikethrough, ada link-nya, bisa kita perbesar huruf-nya, bisa pakai list, dst.

Dan yang penting, cara makainya gampang banget.

<https://github.com/basecamp/trix?tab=readme-ov-file#getting-started>

Pertama, tambahkan baris berikut di atas di bagian <head> kita pada `resources/views/dashboard/layouts/main.blade.php`

```
<head>
...
<link rel="stylesheet" type="text/css"
      href="https://unpkg.com/trix@2.0.8/dist/trix.css">
<script type="text/javascript"
      src="https://unpkg.com/trix@2.0.8/dist/trix.umd.min.js"></script>
</head>
```

* Bisa pakai cdn atau kita download source code-nya (zip)

Download source-code (zip-nya)

Pindahkan file `/dist/trix.css` dan `/dist/trix.js` ke `/public`

* Namun folder dist sudah tidak ada sekarang, jadi pakai CDN aja.

Kemudian kita tambahkan

<https://github.com/basecamp/trix#integrating-with-forms>

```
<form ...>
  <input id="x" type="hidden" name="content">
  <trix-editor input="x"></trix-editor>
</form>
```

Menjadi seperti ini

```
<div class="mb-3">
  <label for="body" class="form-label">Body</label>
  <input id="body" type="hidden" name="body">
  <trix-editor input="body"></trix-editor>
</div>
```

input id disamakan, supaya apa yang kita tulis di `<trix-editor>` itu masuk ke dalam input-nya. Terus dikirim ke Controller-nya.

Kemudian saat kita cek pada halaman create post kita sudah ada si editor-nya.

Keren banget!

Menon-aktifkan fitur upload image pada Trix Editor

Kita hilangkan tombol untuk file upload-nya. Supaya orang nanti tidak bisa iseng buat nambahin file upload.

Kita harus melakukan 2 hal, kita ilangin dari sisi css-nya dan kita akan aktifkan fungsi nya di javascript-nya.

```
<style>
  trix-toolbar [data-trix-button-group="file-tools"] {
    display: none;
  }
</style>
```

Berikutnya, agar ngga jalan fiturnya kita bisa kasih javascript dibawah.

```
document.addEventListener('trix-file-accept', function(e) {
  e.preventDefault();
})
```

Mencoba mengirim data dari form

Sekarang kita cobain isi dulu data-nya, kita belum ngasih validasi apa-apa. Kita cobain aja data-nya kekirim apa ngga

Nah dari form kita itu kan dikirim ke **/dashboard/posts** kemudian di teruskan ke method store()
kita coba return request buat nangkep data yang dikirimin

```
public function store(Request $request)
{
  // untuk menjalankan fungsi tambah-nya
  return $request;
}
```

Kemudian kita coba isikan form-nya.

Kita submit
Dan pastikan data-nya terkirim ke halaman store-nya.

[Finish]

19. Validation & Insert Post

Validasi form create

Kita kasih validasi dulu sebelum kita insert.

DashboardPostController

```
public function store(Request $request)
{
    $validatedData = $request->validate([
        'title' => 'required|max:255',
        'slug' => 'required|unique:posts',
        'category_id' => 'required',
        'body' => 'required'
    ]);
}
```

Kemudian kita coba kirim form kosongan, harusnya tidak terkirim ke method store()

Menampilkan pesan error

Kita akan kirim pesan kesalahannya.

Seperti biasa, kita akan bikin supaya setiap input-nya itu isInvalid

Kita ke halaman view create

resources > views > dashboard > posts > create.blade.php

```
<div class="mb-3">
    <label for="title" class="form-label">Title</label>
    <input type="text" class="form-control @error('title')
is-invalid @enderror" id="title" name="title">

    @error('title')
```

```
<div class="invalid-feedback">
    {{ $message }}
</div>
@enderror

</div>
```

Kita lakukan hal yang sama juga untuk **input form Slug**

Untuk yang **category**, sebetulnya kita tidak perlu melakukan itu. dan sebetulnya kita tidak perlu melakukan validasi. Tapi supaya nanti data di **\$validateData** -nya masuk. Jadi kita simpan disitu.

Terus juga untuk yang **body**. Karena yang body ini adalah editor yang bukan milik-nya bootstrap, jadi kita tidak bisa kasih error **class is-invalid**.

Tapi kalau mau, kita bisa tambahkan di bawah label-nya sebuah alert dari bootstrap. Atau bisa kita kasih tulisan biasa aja.

```
<div class="mb-3">
    <label for="body" class="form-label">Body</label>

    @error('body')
        <p class="text-danger">{{ $message }}</p>
    @enderror

    <input id="body" type="hidden" name="body">
    <trix-editor input="body"></trix-editor>
</div>
```

Kalau kita mau menanganiinya lewat html, kita bisa tambahkan **<required>** Kita tambahkan juga **<autofocus>** pada input form title, agar supaya otomatis fokus.

Menampilkan inputan lama

Kalau kita sudah mengisikan semua form, kemudian ada yang kelupaan belum diisi. Jadi yang sudah kita isikan akan hilang.

Agar tidak hilang, kita tambahkan

```
value="{{ old('title') }}"
```

Kita terapkan juga untuk yang Slug.

Untuk yang **Category** kita buat agar opsi yang sudah kita pilih, akan tetap.

```
<select class="form-select" name="category_id">

    @foreach ($categories as $category)

        @if ( old('category_id') == $category->id)
            <option value="{{ $category->id }}" selected>{{ $category->name }}</option>
        @else
            <option value="{{ $category->id }}">{{ $category->name }}</option>
        @endif

    @endforeach

</select>
```

* Kita pakai **==** bukan **==** agar tidak mengecek tipe data-nya juga, karena yang satu tipe-nya **string** dan satunya **integer**

```
@if ( old('category_id') == $category->id)
```

Untuk yang **body**, kita juga bisa samakan dengan title dan slug dengan menambahkan value pada <input body>.

Menyimpan data ke tabel post

Selanjutnya kita coba insert data-nya.

Kita buka lagi controllernya **DashboardPostController**

Ingat, kita baru punya 4 data yang sudah validasi. Kita butuh data yang lain sebelum bisa kita insert.

Kita akan bikin dulu data baru

```
$validatedData['user_id'] = auth()->user()->id;
```

Excerpt ini kita ngambil dari **body**. jadi kalau body-nya banyak akan kita limit misalnya hanya 100 karakter saja.
(saran dari teman-teman)

Dokumentasi Str Limit :

<https://laravel.com/docs/8.x/helpers#method-str-limit>

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog',
20, ' (...)');

// The quick brown fox (...)
```

Kita terapkan ke web kita.

```
use Illuminate\Support\Str;

$validatedData['excerpt'] = Str::limit($request->body, 200);
```

Nah, data **excerpt** yang dikirim dari **body** format data-nya dari **editor trix**, jadi ada tag-tag HTML-nya.

Supaya tidak ada tag HTML-nya, kita bisa pakai fungsi-nya PHP yang namanya **strip_tags()**

```
use Illuminate\Support\Str;

$validatedData['excerpt'] = Str::limit(strip_tags($request->body), 200);
```

Kalau sudah, langsung aja kita insert.

```
Post::create($validatedData);
```

Menampilkan Pesan Sukses

Setelah itu kita balikin lagi ke halaman Post.
Sambil kita kirimin pesan sukses.

```
return redirect('dashboard/posts')->with('success', 'New Post has  
been added!');
```

Kemudian kita tambahkan alert-nya di index dashboard post
resources > views > dashboard > posts > index.blade.php

<https://getbootstrap.com/docs/5.0/components/alerts/>

```
@if ($session->has('success'))
    <div class="alert alert-success col-lg-8" role="alert">
        {{ $session('success') }}
    </div>
@endif
```

* Note : Ternyata jika form input-nya di disable, tidak bisa ngirim.

Memperbaiki bagian lain

Menghapus back pada show dashboard posts

resources > views > dashboard > posts > show.blade.php

```
{-- <a href="/posts" class="d-block mt-3">Back To Posts </a> --}
```

Menambah margin pada creat posts

resources > views > dashboard > posts > create.blade.php

```
<form method="POST" action="/dashboard/posts" class="mb-5">
```

* Jika sudah ada slug yang sama, maka sluggable akan otomatis membuat unique, namun jika masih kekeh untuk mengganti, maka akan di jagain sama laravel-nya agar tidak ada slug yang sama.

*

Fitur keamanan trix editor

Kalau kita menggunakan editor seperti trix editor, yang didalamnya bisa nulisin tag HTML.

Bagaimana kalau ada user jail yang nyimpen script jahat di dalamnya?

Nah kita ngga perlu khawatir, hal tersebut sudah ditangani oleh editornya.

semisal kita sisipkan script javascript

```
<script>alert('hahaha');</script>
```

Dia sudah otomatis dilakukan escape karakter.

[Finish]

20. Update & Delete Post

Membuat fitur delete

Kita akan mengubah link untuk hapus, menjadi sebuah tombol di dalam form. Fitur hapus ini harus kita lakukan ke dalam form karena kita butuh REQUEST method-nya delete dan juga fitur CSRF

resources > views > dashboard > posts > index.blade.php

```
<form action="/dashboard/posts/{{ $post->slug }}"
      method="POST" class="d-inline">
    @method('delete')
    @csrf

    <button class="badge bg-danger border-0"
           onclick="return confirm('Are you sure?')">
      <span data-feather="x-circle"></span>
    </button>

</form>
```

Jika kita jalankan tombolnya, maka akan muncul error seperti ini, karena kita masih belum menangani route-nya

Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException
The DELETE method is not supported for this route. Supported methods: GET, HEAD, POST.

<http://127.0.0.1:8000/dashboard/posts>

Kita buka lagi
DashboardPostController

Ini simple banget!

Sebenarnya kita cuman butuh 2 baris yang ada di method create(), tapi nanti kita ganti dengan destroy()

```
public function destroy(Post $post)
{
    Post::destroy($post->id);
    return redirect('dashboard/posts')->with('success', 'Post
has been deleted!');
}
```

Dan sekarang kita sudah bisa menghapus data dari halaman index dashboard posts.

Menghapus postingan dari halaman detail (show)

Kita tinggal meng-copy kan dari yang sebelumnya ke show
resources > views > dashboard > posts > show.blade.php

Membuat halaman update

Halaman update mirip dengan halaman create, namun beda-nya sudah ada isinya. Nanti kita akan contek dari view create.

Kita masuk ke halaman view index posts dulu, kita arahkan link-nya tombol edit ke halaman edit

```
href="/dashboard/posts/{{ $post->slug }}/edit"
```

* sama seperti link ke create, namun di kasih tambahan **/edit**
Ini aturan default dari si **route resource-nya**

Kalau kita pengen tau **route resource** kita itu ada apa aja, kita bisa lihat lewat terminal

```
$ php artisan route:list
```

Jadi kita bisa lihat daftar dari route kita.

Seluruh route kita ada di tabel tersebut, baik yang kita tulis manual maupun yang di generate oleh laravel-nya.

kalau mau EDIT

```
| GET|HEAD | dashboard/posts/{post}/edit
```

Misal nanti mau UPDATE

method-nya kalo ngga PUT, pakai PATCH

```
| PUT|PATCH | dashboard/posts/{post}
```

kalau mau DELETE, **method-nya** DELETE

```
| DELETE | dashboard/posts/{post}
```

Jadi rapi, semua sama depannya, dashboard/posts.. yang membedakan hanya request method-nya.

Sekarang tinggal kita arahkan ke view-nya. caranya, kita cukup duplikat aja view create.

Kita ganti namanya jadi **edit.blade.php**

resources > views > dashboard > posts > edit.blade.php

Tinggal kita benerin di Controller-nya

Kita cari method edit ()

edit() untuk nampilin view-nya, **update()** untuk proses ubah-nya.
sama seperti tambah data kita. **create()** untuk nampilin view-nya, **store()** untuk proses datanya.

DashboardPostController

```
public function edit(Post $post)
{
    // halaman buat nampilin ubah data
    return view('dashboard.posts.edit', [
        'post' => $post,
        'categories' => Category::all()
    ]);
}
```

* Adadata yag harus kita kirim, yaitu data post-nya. Supaya nanti kita bisa isi.
Kita akan isi post-nya, ngambil dari (**Post \$post**)
sudah di kasih **route model binding**, tinggal kita kasih '**post' => \$post**
(tinggal kita pakai di edit-nya nanti)

resources > views > dashboard > posts > edit.blade.php

```
<h1 class="h2">Edit Post</h1>
```

```
<form method="POST" action="/dashboard/posts/{{ $post->slug }}">
    @method('put')

    {{-- atau bisa pakai @method('patch') --} }

    @csrf
```

Cara ngisi nilai-nilai-nya input.
Kan kita punya **value old()**

Nah value old() ini nanti kalo kita udah masuk ke validasi. dan kalo validasinya ngga lolos, nanti isinya tetap nilai lama-nya.

Nah laravel punya fitur yang keren.
Kalau misalkan kita tambahkan seperti ini.

```
value="{{ old('title', $post->title) }}"
```

maka nanti laravel-nya ngecek, ada **old()** ngga?
Apakah ada old() ? kalo ngga ada, tampilin isi dari **\$post->title**

Kita lakukan yang sama untuk yang slug

Nah, untuk yang category sama. Tidak perlu repot-repot kasih IF lagi. tinggal tambahan seperti ini

```
@if ( old('category_id', $post->category_id) == $category->id)
```

Untuk yang body, kita tambahin di input type hidden-nya.

```
<input id="body" type="hidden" name="body" value="{{ old('body', $post->body) }}">
<trix-editor input="body"></trix-editor>
```

* karena trix editor nya sudah ngecek pakai javascript apa isi dari input type hidden-nya.

Memperbaiki validasi slug

Nah, sekarang...

Sebelum kita jalankan update post-nya dan juga validasi-nya, ada yang harus kita pikirin dulu. Yaitu adalah **SLUG**

SLUG ini agak **tricky**... Karena slug ini unique (tidak boleh ada yang sama). Kalau misalnya kita mau ubah hanya category-nya aja (sisanya sama semua), nanti begitu kita update, maka SLUG ini akan di update juga. Begitu di update, ngga akan bisa. Karena **SLUG-nya sama**. Kecuali kita ubah slug-nya menjadi slug yang baru. Nah baru mau...

Nanti nih harus kita akalin supaya ngga begitu aja ngubah, nanti kita lihat validasinya.

oke, tapi pahami dulu kita punya problem itu.

Sekarang kita balik lagi ke controller kita.

DashboardPostController

Kita akan isi method update() nya.

Kita akan lakukan validasi lagi.

```
$validatedData = $request->validate([
    'title' => 'required|max:255',
    'slug' => 'required|unique:posts',
    'category_id' => 'required',
    'body' => 'required'
]);
```

* Dari validasi yang kita copy dari **method store()** terdapat validasi untuk SLUG, yang isi nya unique dari tabel posts.

Tapi problemnya, gimana kalo slug-nya itu sama (tidak diubah)?
nanti akan problem nih, karena slug yang lama itu udah ada di dalam database.
Maka dari itu validasi untuk slug kita keluarin. Kita akan cek menggunakan kondisi.

Kondisinya seperti ini :

Kalau misalnya slug yang baru, itu sama dengan slug yang lama. Berarti jangan di kasih validasi (biarin lolos!).

Tapi kalau beda atau berubah (slug-nya ganti manual/judulnya ganti), kita validasi

```
public function update(Request $request, Post $post)
{
    // halaman untuk proses ubah data-nya
}
```

* slug yang baru/ data yang baru kita ngambilnya dari **\$request** (apa yang kita tulis yang baru)

* **\$post** adalah data lama yang ada di tabel kita.

```
public function update(Request $request, Post $post)
{
    $rules = [
        'title' => 'required|max:255',
        // 'slug' => 'required|unique:posts',
        'category_id' => 'required',
        'body' => 'required'
    ];

    if($request->slug != $post->slug) {
        $rules['slug'] = 'required|unique:posts';
    }
}
```

```
    $validatedData = $request->validate($rules);
}
```

* kalau kosong juga bakalan di validasi.

Mengubah data

Selanjutnya kita akan lakukan update()

Dokumentasi Mass Updates :

<https://laravel.com/docs/8.x/eloquent#mass-updates>

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are active and have a destination of San Diego will be marked as delayed:

```
Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

* where ID sama dengan berapa, terus update()
(ini yang akan kita lakukan)

Atau ada lagi cara lainnya kita bisa pakai updateOrCreate()

Dokumentasi upserts :

<https://laravel.com/docs/8.x/eloquent#upserts>

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Kita tambahkan setelah validasi seperti ini

```
$validatedData = $request->validate($rules);

$validatedData['user_id'] = auth()->user()->id;
$validatedData['excerpt'] =
Str::limit(strip_tags($request->body), 200);

Post::where('id', $post->id)
->update($validatedData);

return redirect('dashboard/posts')->with('success', 'New Post has
been updated!');
```

Kemudian kita bisa coba edit postingan di web kita.

Kita update juga link tombol edit pada **show.blade.php** kita.

```
href="/dashboard/posts/{{ $post->slug }}/edit"
```

* kita pastikan lagi jika kita ubah slug-nya manual, jika ada yang sama dengan postingan yang lain maka tidak bisa

[Finish]

21. Upload Image

Membuat input file

Sebelumnya postingan kita sudah ada gambar yang diambil dari API-nya unsplash

Dan didalam tabel kita juga masih belum ada field untuk menyimpan gambar.

Idenya, nanti untuk postingan yang ngga ada gambarnya, kita akan pakai gambar yang diambil dari API-nya unsplash

Kita akan tambahkan **inputan baru** sebelum body pada halaman **create new post** untuk menyimpan/mengupload file. Nanti file-nya akan kita validasi, yang boleh di upload hanya gambar saja.

untuk ngambil file input-nya, kita akan pakai file input dari bootstraps

Dokumentasi File input :

<https://getbootstrap.com/docs/5.0/forms/form-control/#file-input>

resources > views > dashboard > posts > create.blade.php

```
<div class="mb-3">
    <label for="image" class="form-label">Post Image</label>
    <input class="form-control" type="file" id="image"
name="image">
</div>
```

Perlu di ingat !!!

ketika kita mau bekerja dengan file di dalam form kita, kita harus ubah dulu form-nya.

Kita harus tambahkan sebuah **atribut** yang nama-nya **enctype**. Karena kalau ngga, form-nya tidak bisa menangani file. kita kasih namanya “**multipart/form-data**”

```
<form method="POST" action="/dashboard/posts" class="mb-5"
enctype="multipart/form-data">

@csrf
```

Sehingga form-nya bisa menangani 2 hal, yang pertama : semua inputan dalam bentuk **text** akan diambil pakai **request biasa**. Lalu kalau ada **file**, akan di ambil menggunakan **REQUEST FILE**

Kalau tidak ada **enctype="multipart/form-data"** , maka file kita tidak akan bisa di upload.

Cara kerja proses upload

Sekarang, sebelum kita jalanin fungsi upload-nya.

Kita akan coba lihat dulu, bagaimana **laravel menangani file itu seperti apa?!**

Kita buka dulu **DashboardPostController**

Kita cari **method store()**

Dari pada scroll-scroll, kita bisa **Ctrl + P** > tulis **@store**

di **method store()**

Jadi kita tau, kalau misalkan **\$request**, itu menangani semua data yang dikirimin dari form.

Kita coba **DDD** (Dumb, Die, and Debug) **\$request**
kita akan **var_dump** sambil **nge-debug**

```
public function store(Request $request)
{
    ddd($request);

    // script dibawahnya tidak akan dijalankan.

}
```

Kemudian kita coba input form di halaman web Create New Post, dan kita coba upload gambar (atau file apa saja). kemudian kita submit

Kemudian kita akan diarahin ke **Dumb, Die, and Debug**-nya



The screenshot shows a terminal window with the following output:

```
info /home/fafaiez/Documents/coding/LARAVEL/coba-laravel/
Dump, Die, Debug
http://127.0.0.1:8000/dashboard/posts
```

Banyak banget data yang dikirim.

Yang kita butuhin cuman yang ada di dalam [**+request**]

```
Illuminate\Http\Request {#45 ▾
  +attributes: Symfony... \ParameterBag {#47 ▶ }
  +request: Symfony... \InputBag {#46 ▾
    #parameters: array:5 [▼
      "_token" => "xRshcchEAdIEhxojNkV7F7R1GKMHE1UehAaoMZGd"
      "title" => "lorem"
      "slug" => "lorem"
      "category_id" => "1"
      "body" => null
    ]
  }
}
```

Untuk gambar-nya ada di dalam [**+files**]

Maka dari itu kita butuh “**multipart/form-data**”

Sehingga yang string masuknya ke [**+request**], yang **file** masuknya ke [**+files**]

Dan di dalam [**+files**] akan banyak data-nya. mulai dari nama file aslinya, ukuran file-nya, dll.

```
+query: Symfony...\InputBag {#53 ▶ }
+server: Symfony...\ServerBag {#50 ▶ }
+files: Symfony...\FileBag {#49 ▶ }
#parameters: array:1 [▼
    "image" => Symfony...\UploadedFile {#34 ▶
        -test: false
        -originalName: "Screenshot from 2024-12-31 11-13-08.png"
        -mimeType: "image/png"
        -error: 0
        path: "/tmp"
        filename: "php1qhqtz"
        basename: "php1qhqtz"
        pathname: "/tmp/php1qhqtz"
        extension: ""
        realpath: "/tmp/php1qhqtz"
        aTime: 2024-12-31 08:39:25
        mTime: 2024-12-31 08:39:25
        cTime: 2024-12-31 08:39:25
        inode: 170
        size: 171438
        perms: 0100600
        owner: 1000
        group: 1000
        type: "file"
        writable: true
        readable: true
        executable: false
        file: true
        dir: false
        link: false
    }
]
}
```

Dan file ini yang akan kita simpan.

Sekarang, gimana nyimpen nya?

cara nyimpen-nya, sebenarnya gampang banget.
Kita cukup tulis kayak gini.

script yang di bawah **RETURN** tidak akan dijalankan, karena sudah return. Jadi abaikan script yang ada di bawahnya.

```
public function store(Request $request)
{
    return $request->file('image_post')->store('post-images');
```

```
//file (apapun) bernama 'image_post' akan di simpan di folder  
'post-images'  
  
    // script dibawahnya tidak akan dijalankan.  
  
}
```

Kemudian kita coba lagi input form di halaman web Create New Post, dan kita coba upload gambar (atau file apa saja). kemudian kita submit

Kemudian **fungsi store()** akan otomatis mengembalikan **PATH-nya** (selain mengupload/menyimpan file-nya)

dan file-nya sudah tersimpan di folder “**post-images**” yang otomatis dibuatkan, dengan nama random.

storage > app > post-images > namafilerandom.jpg

Keren yaa! dengan 1 baris doang, kita sudah bisa upload file. Cuman masih banyak yang harus di perbaiki. Tapi terlepas dari itu ini AJAIB banget!

Konfigurasi file storage

Apa yang harus kita perbaiki?

Kita akan atur dulu tempat penyimpanan-nya.

sebelum itu, gimana kalau kita lihat dokumentasinya dulu mengenai **FILE STORAGE**

Dokumentasi **File System/File Storage** :

<https://laravel.com/docs/8.x/filesystem#introduction>

Laravel provides a powerful filesystem abstraction thanks to the wonderful [Flysystem](#) PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3. Even better, it's amazingly simple to switch between these storage options between your local development machine and production server as the API remains the same for each system.

Si laravel sudah menyediakan sebuah sistem yang powerfull gara-gara ada library/package yang namanya **Flysystem**. Jadi nanti kedepannya kalo misalkan kita mau integrasikan aplikasi kita agar bisa ngupload ke beberapa tempat misalnya di local (komputer kita), mau lewat FTP di server yg punya kita sendiri, atau bahkan lewat Amazon S3 (di tempat penyimpanan terpisah). Atau mau sekaligus ketiga-tiganya bisa.

Cuman nanti yang akan kita lakukan ke local dulu aja.

Nah gimana cara ngaturnya?

ada di bagian konfigurasi file system

`config/filesystems.php`

Laravel's filesystem configuration file is located at `config/filesystems.php`

config > filesystem.php

```
| Default Filesystem Disk  
  
'default' => env('FILESYSTEM_DRIVER', 'local'),
```

* Secara **default** filesystem disk-nya adalah '**local**'

* akan ngecek dulu di ENVIRONMENT Variabel-nya atau ngga, klo ngga ada pilih local

```
'disks' => [  
  
    'local' => [  
        'driver' => 'local',  
        'root' => storage_path('app'),  
    ],  
  
],
```

* nah, local itu ada didalam folder ‘storage’, di dalam folder ‘app’

Maka dari itu percobaan upload yang sebelumnya, file-nya akan tersimpan di folder “post-images” yang otomatis dibuatkan, dengan nama random.
storage > app > post-images > namafilerandom.jpg

Kedepannya kita ngga mau nyimpen disitu, karena kita pengen agar file-file yang di upload itu bisa **diakses secara public**, karena kita pengen nampilin di halaman blog kita.

Jadi nanti kita harus pindahin di **public**
storage > app > public

maka dari itu default filesystem-nya jangan “**local**”, kita bisa pindahin ke “**public**”

Salah satu caranya, kita bisa ubah default local-nya ke public seperti ini.

```
| Default Filesystem Disk  
  
'default' => env('FILESYSTEM_DRIVER', 'public'),
```

Tapi pak dhika biasanya lebih suka pindahin ke **environment variable (.env)**

Jadi tambahkan ini di **.env**

```
FILESYSTEM_DRIVER=public
```

Sekarang kita coba upload gambar sekali lagi.

dan file-nya akan tersimpan di folder “**public**”
storage > app > public > post-images > namafilerandom.jpg

Tapi ini masih ada problem !!!

Kalau misalkan kita mau coba akses langsung gambarnya lewat browser, data nya tidak akan tampil (**404 Not Found**)

Copy Relative Path

<http://127.0.0.1:8000/storage/app/public/post-image/namafilerandom.png>

Jadi kalau kita upload di **storage > app > public** , dan ingin tampilkan di blog kita.. itu tidak akan tampil karena tidak bisa diakses.

Loh kok ngga bisa di akses pak? katanya public pak??

Nah ini gara-gara **folder “public”** yang ada didalam **folder “storage”**
storage > app > public

itu harus kita hubungkan dulu dengan **folder “public”** yang ada didalam aplikasi kita

belajar-laravel > public

dan folder ini adalah folder yang bener-bener bisa di akses oleh user

dan bisa di akses di browser dan bisa di akses oleh siapapun.

public > img > namafilegambar.png

<http://127.0.0.1:8000/img/namafilegambar.png>

Kalau folder **storage > app > public > post-images > namafilerandom.jpg** tidak bisa

Bagaimana cara menghubungkan antara **folder “public”** yang di **“storage”** dengan **folder “public”** yan ada di aplikasi kita?

Kita harus gunakan yang nama-nya **Symbolic Link**

Dokumentasi

<https://laravel.com/docs/8.x/filesystem#the-public-disk>

Cara nya kita tinggal tulis di terminal kita

```
$ php artisan storage:link
```

Setelah itu akan ada **folder “storage”** yang ada panahnya (symbolic link) di dalam folder “public” apikasi kita

Dan sekarang bisa di akses di browser dan bisa diakses oleh siapapun.

<http://127.0.0.1:8000/storage/app/public/post-image/namafilerandom.png>

Dan sekarang, kita sudah siap mengakses file-nya

Dan di dokumentasi, untuk memanggilnya menggunakan **asset()**

```
echo asset('storage/file.txt');
```

Membuat field image di tabel post

Sebelum kita menjalankan upload file-nya,
Ingat! di dalam database kita, kita tidak punya **field image**.

nah, gimana kita mau naruh gambarnya kalo gitu.
maka dari itu kita bikin dulu di migrasi kita

Jadi mau ngga mau database-nya harus kembali kita bersihkan. tapi harusnya sekarang udah ngga ada masalah, karena kita udah tau **fitur sakti** yang namanya **migration** dan juga **factory**.

buka file migrasi posts table-nya

database > migrations > 2024_04_13_143059_create_posts_table.php

tambahkan field **image** di setelah slug

```
$table->string('image')->nullable();
```

* kita pakai **string**, karena kita mau nyimpen nama dan tempat penyimpanan-nya aja.

* boleh kosong (**nullable**), karena kita akan pakai gambar dari unsplash (atau yang lain)

Sebelum kita jalanin migrasinya, kita buka Seeder-nya dulu.

database > seeders > DatabaseSeeder.php

tetap seperti sebelumnya, bikin User, category, dan factory untuk isi post-nya

tapi kita buat akun user untuk kita sendiri, tambahkan username

Kemudian jalankan

```
$ php artisan migrate:fresh --seed
```

Dan sudah di reset lagi semua database-nya.

Memperbaiki validasi email

jika kita menggunakan email dari seeder-nya akan di generate
lorem ipsum@example.org

kalau kita login dengan email itu, maka tidak bisa (email tidak valid)

dan itu salah di validasi kita, **validasi kita terlalu ketat.**

Buka **LoginController.php**

kita ubah di bagian **authenticate()**

```
'email' => 'required|email:dns',
```

kita bisa **hilangkan :dns**

agar email @example.org bisa masuk.

Validasi image

Sekarang kita jalanin validasinya untuk input gambar.

DashboardPostController

Tambahkan validasi untuk image, di method store()

```
'image_post' => 'image|file|max:1024', //maximal 1 MB
```

* validasi “**image**” hanya untuk file nya harus berupa gambar

* untuk validasi ukuran harus di awali dengan **|file|**

kemudian validasi-nya apa. semisal **max, min, size** (harus persis berapa ukurannya)

Karena kalau ngga pakai `|file|`, maka akan di anggapnya sebagai karakter.
kalau dengan `|file|`, maka di anggapnya KiloByte (KB).

lebih detailnya googling lihat dokumentasi.

Kita cek validasi-nya.
kita akan kasih klo gagal gimana?!

tambahkan `@error('image_post')`

```
<input class="form-control @error('image_post') is-invalid  
@enderror" type="file" id="image" name="image_post">  
</div>  
  
@error('image_post')  
    <div class="invalid-feedback">  
        {{ $message }}  
    </div>  
@enderror
```

Nah, teruss.. salah satu problem dari image adalah kita ngga bisa ngasih `old()` karena pertimbangan **security**.

Jadi jangan sampai nanti orang bisa tau struktur directory kita. maka dari itu `old()` tidak bisa diterapkan.

Kemudian, kita coba upload gambar yang ukurannya besar di atas 1 MB atau upload file yang bukan gambar (semisal PDF)
Maka akan gagal.

Kita hanya ngejagain ukuran file-nya saja, tidak kita lakukan re-size untuk ukuran gambarnya

Kemudian, sekarang yang akan kita lakukan adalah kita akan cek dulu, gimana klo misalnya user ngga ngisiin gambarnya (kan gapapa, ngga wajib), karena klo gambarnya kosong, maka akan di ambil dari unsplash (atau dari kita sendiri).

DashboardPostController

Tambahkan kondisi untuk image, di method store()

```
if($request->file('image_post')) {  
    $validatedData['image_post'] =  
$request->file('image_post')->store('post-images');  
}  
  
// kalau request dari file yang namanya 'image_post' itu ada isinya (true),  
// maka kita akan nambahin 1 buah $validatedData lagi (image), yang di isi  
dengan uplaoad gambarnya,  
// sekaligus ambil nama gambarnya. kemudian kita store()
```

Kemudian kita coba bikin postingan baru.

Kita pastikan gambarnya sudah ada di folder **storage > post-image**

dan pastikan pada kolom image juga terisi nama folder dan nama file gambarnya.
post-images/namafilerandom.jpg

Menampilkan image pada post

Kita buat pengecekan, jika gambarnya ada maka diambil dari database. jika tidak ada, maka di ambil dari unsplash (disini dari gambar sendiri)

Kita benerin dulu show yang ada di halaman dashboard

Update **resources > views > dashboard > posts > show.blade.php**

```
@if ($post->image)  
  
{{-- <div style="max-height: 350px; overflow:hidden;"> --}}
```

```

        category->name }}">
    {{-- </div> --} }

    @else

        category->name }}">

    @endif

```

Sekarang kita mesti benerin yang ada di halaman depannya atau halaman frontend-nya (user)

Update resources > views > posts.blade.php

```

@if ($posts->count() > 0)
    {{-- kondisi jika true --}}
    <div class="card mb-3">

        @if ($posts[0]->image)
            <div style="max-height: 350px; overflow:hidden;">
                category->name }}">
            </div>
        @else
            category->name }}">
        @endif
    </div>

```

Dan itu untuk gambar postingan paling terbaru,
sekarang untuk potingan no.2 dari yang terbaru

Kita coba bikin postingan baru lagi

Update juga untuk gambar yang ada di posisi urutan kedua di
resources > views > posts.blade.php
dengan cara yang sama

Dan terakhir kita juga update untuk halaman single post-nya
resources > views > post.blade.php
sama persis dengan yang **show.blade.php**

[Finish]

Selasa, 04 Februari 2025 (00.25)

22. Preview, Update & Delete Image

Membuat fitur preview image

Pada saat kita pilih gambar, gambarnya akan tampil preview-nya.

Caranya kita butuh Javascript.

Dan akan jalan di browser-browser yang terbaru, kalau browsernya versi lama seperti Internet Explorer tidak akan jalan.

Kita tambahkan seperti berikut pada create post

resources > views > dashboard > posts > create.blade.php

```
<input class="form-control @error('image_post') is-invalid @enderror" type="file" id="image_post" name="image_post" onchange="previewImage()">
```

* Di dalam input image, akan kita kasih sebuah EVENT yang namanya **ONCHANGE**

* ketika gambarnya berubah (atau yang tadinya kosong, jadi ada isinya), kita akan panggil sebuah FUNCTION di JavaScript yang namanya “`previewImage()`”

Yang nantinya, fungsi tersebut akan menampilkan gambar untuk preview-nya.

```
<img class="img-preview img-fluid mb-3 col-sm-5">
```

* Ada tag gambar yang tidak ada SRC-nya

Kemudian kita bikin fungsi JavaScript-nya di bawah.

```
function previewImage() {
    const image = document.querySelector('#image_post');
    const imgPreview = document.querySelector('.img-preview');

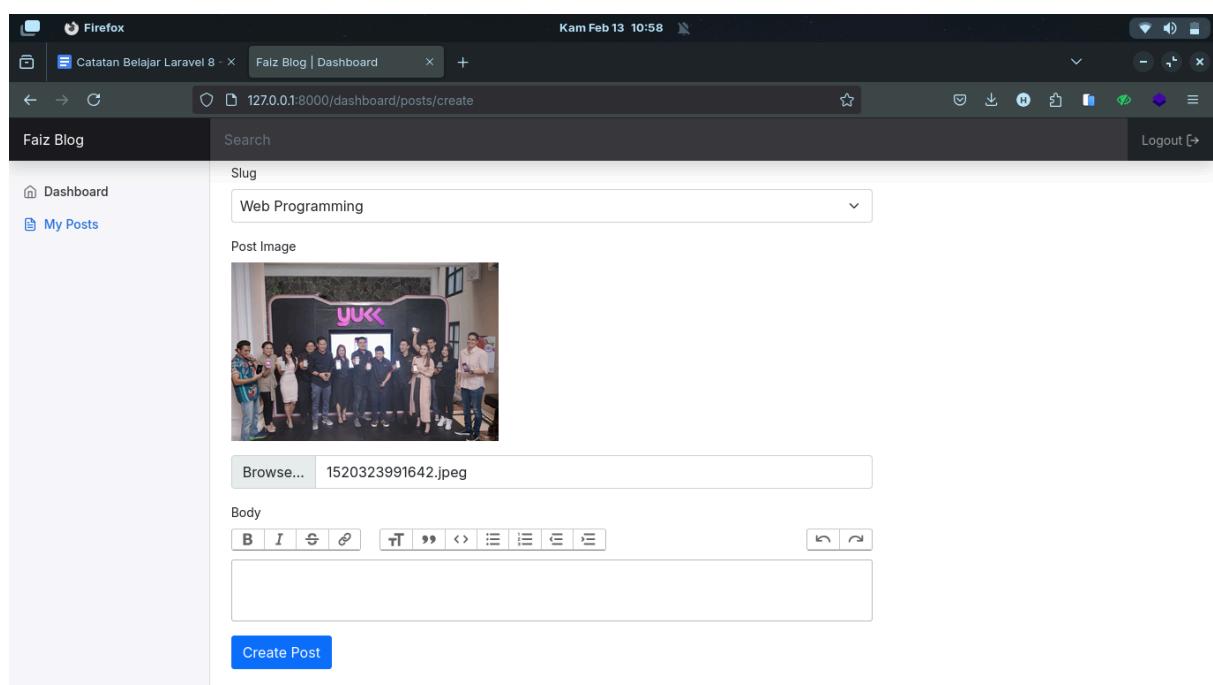
    imgPreview.style.display = 'block';
```

```
const oFReader = new FileReader();
oFReader.readAsDataURL(image.files[0]);

oFReader.onload = function(oFREvent) {
    imgPreview.src = oFREvent.target.result;
}

}
```

Nah, kalau berhasil.. Sekarang preview image sudah tampil.



Update image

Kita update seperti berikut pada edit post

resources > views > dashboard > posts > edit.blade.php

Pada FORM-nya kita kasih **enctype="multipart/form-data"**

```
<form method="POST" action="/dashboard/posts/{{ $post->slug }}"
class="mb-5" enctype="multipart/form-data">
    @method('put')
```

Tambahkan INPUT IMAGE seperti di create post (**create.blade.php**)
dan juga script (JavaScript) fungsi **previewImage()**

Menampilkan preview image lama (yang sudah ada)

Nah sekarang, jika kita cek di website-nya. Sudah terdapat input untuk edit gambaranya.

Namun preview gambarnya (yang sudah ada sebelumnya) tidak ditampilkan. Seharusnya kalau sudah ada gambar langsung di tampilkan gambarnya.

Gimana caranya?

Maka dari itu kita butuh kondisi. Seperti berikut!

```
@if ($post->image)

@else

    <img class="img-preview img-fluid mb-3 col-sm-5">

@endif
```

Dan sekarang, pada halaman edit, sudah muncul gambar yang lama, dan juga sudah bisa mengganti dengan gambar yang baru.

Namun kita masih belum bisa melakukan **SUBMIT**, karena di method **UPDATE()** kita masih belum ada **VALIDASI**-nya.

Menambahkan validasi image untuk update()

Kita tambahkan validasi image di method **UPDATE()**
DashboardPostController

```
public function update(Request $request, Post $post)
{
    // halaman untuk proses ubah data-nya
    $rules = [
        'title' => 'required|max:255',
        'category_id' => 'required',

        'image' => 'image|file|max:1024', //maximal 1 MB

        'body' => 'required'
    ];
    ...
}
```

Terus, kita juga butuh cek ada image baru apa ngga?!

Kalau ada image baru upload,
Kalau tidak ada, ya udah pakai image lama.
atau kalau kosong, yaudah biarkan kosong.

Jadi kita tambahkan setelah validasi-nya sudah lolos

```
$validatedData = $request->validate($rules);

$validatedData['user_id'] = auth()->user()->id;
```

```
$validatedData['excerpt'] =  
Str::limit(strip_tags($request->body), 200);  
  
// upload foto setelah validasinya lolos  
if($request->file('image_post')) {  
    $validatedData['image'] =  
$request->file('image_post')->store('post-images');  
}  
...  
...
```

* Jika upload foto-nya diletakkan sebelum validasinya lolos, maka yang masuk ke dalam database kita adalah folder tempat penyimpanan sementara.

Nah, sekarang gambarnya sudah bisa di EDIT, namun **ada sedikit masalah**. Masalahnya adalah gambarnya kita sekarang NUMPUK, padahal ada gambar yang ngga di pake.

Menghapus gambar saat mengupdate gambar baru

Nah, gimana kalo pada saat meng-UPDATE gambar, sekalian kita hapus gambar sebelumnya. supaya ngga menuh-menuhin database.

* Gambar-gambar yang ada di storage bisa di hapus saja (yang tidak di pakai)

Sekarang, cara menghapus-nya kita simpan di dalam IF-nya

* Kalau misalkan ada gambar baru, buat nge-ganti gambar yang lama (yang sebelumnya sudah pasti lolos validasi). Kita HAPUS DULU gambar yang lama, baru upload gambar yang baru.

Kita akan tambahkan hapus gambar sebelum upload gambar pada method

UPDATE()
DashboardPostController

```
// upload foto setelah validasinya lolos  
if($request->file('image_post')) {
```

```
// disini kita hapus dulu gambar yang lama disini  
  
// kemudian baru kita upload gambar yang baru  
$validatedData['image'] =  
$request->file('image_post')->store('post-images');  
}
```

Nah tapi GIMANA CARANYA kita dapetin gambar yang lama??

Gimana kalo kita kirimin dulu DATA GAMBAR LAMA-nya.

Kita balik lagi ke halaman edit post

resources > views > dashboard > posts > edit.blade.php

```
{-- INPUT HIDDEN UNTUK IMAGE LAMA --}  
<input type="hidden" name="oldImage" value="{{ $post->image }}">
```

* Sebelum kita tampilkan preview image-nya (setelah label input gambar)
Kita bikin sebuah INPUT yang type-nya HIDDEN yang VALUE-nya berisi NAMA IMAGE LAMA-nya.

Karena mungkin aja gambar LAMA-nya ngga ada, jadi kita bikin pengkondisian

```
{-- INPUT HIDDEN UNTUK IMAGE LAMA --}  
<input type="hidden" name="oldImage" value="{{ $post->image }}">
```

Kita tambahkan fitur hapus-nya

DashboardPostController

```
// disini kita hapus dulu gambar yang lama disini  
if ($request->oldImage) {  
    Storage::delete($request->oldImage);  
}
```

* Kalau gambar LAMA-nya ada, maka akan dilakukan hapus, kalau tidak ada gambar LAMA-nya maka tidak akan dilakukan.

Namun akan error, karena Storage masih belum kita panggil

Kita lihat dulu **namespace**-nya Storage

Dokumentasi

<https://laravel.com/docs/8.x/filesystem#the-local-driver>

Tambahkan NameSpace ini

```
use Illuminate\Support\Facades\Storage;
```

* TAMBAHAN

Ada kemungkinan di tambahan otomatis namespace Storage, namu dari Clockwork. Bisa kita matikan atau dihapus saja. Karena menyebabkan error.

```
// use Clockwork\Storage\Storage;
```

Tambahan sendiri (mengatasi error)

Karena di database kolom “image”, dan input name-nya menggunakan “image_post”.

Jadi kita kecualikan data “image_post” agar tidak dikirimkan ke database, karena sudah ada “image”.

DashboardPostController

```
// Hapus `image_post` agar tidak dikirim ke database
unset($validatedData['image_post']);

Post::where('id', $post->id)
    ->update($validatedData);
```

Kemudian kita coba Edit gambar dari post yang ada, dan kita **PASTIKAN GAMBAR LAMA SUDAH TERHAPUS**.

Pak Dhika said : **MANTAPPPP!!!**

Delete Image

Terakhir temen-temen, kita jalankan DELETE image di **destroy()**
Jadi pada saat Postnya di delete, harusnya gambarnya ikutan ilang.

Gimana caranya??

Caranya tinggal kita cek lagi gambarnya pakai IF seperti sebelumnya.

```
public function destroy(Post $post)
{
    // untuk delete postingannya

    if ($post->image) {
        Storage::delete($post->image);
    }

    Post::destroy($post->id);
    return redirect('dashboard/posts')->with('success', 'Post has
been deleted!');
}
```

Kemudian kita coba Delete postingan yang ada gambar-nya, dan kita **PASTIKAN GAMBAR di storage SUDAH TERHAPUS**.

Dan itulah cara kita **me-MANIPULASI** image untuk Post kita.

[Finish]

Kamis, 20 Februari 2025 (13.23)

23. Authorization

Kita akan membuat fitur terakhir dari aplikasi sistem blog sederhana kita. Yaitu fitur OTORISASI

Sebelumnya, kita sudah membuat fitur autentikasi untuk login dan registrasi, sekarang kita akan membuat agar user yang tadi sudah berhasil registrasi dan juga login, itu memiliki **PERAN YANG BERBEDA**

Contohnya nanti ada USER BIASA dan juga ADMINISTRATOR

Ceritanya, USE CASE untuk aplikasi kita adalah nanti kita akan punya 1 orang ADMIN yang bisa mengelola hal lain selain user biasa.

Contohnya, nanti kita akan coba agar si Admin tersebut dapat mengelola KATEGORI.

Dan sebetulnya, Laravel ini juga mendukung untuk pengelolaan ROLES yang disebutnya POLICIES (yang lebih kompleks lagi), sehingga di dalam aplikasinya kita bisa menentukan banyak roles selain Admin dan User biasa.

Lebih jelasnya kita lihat dokumentasinya saja.

<https://laravel.com/docs/8.x/authorization#introduction>

In addition to providing built-in [authentication](#) services, Laravel also provides a simple way to authorize user actions against a given resource. For example, even though a user is authenticated, they may not be authorized to update or delete certain Eloquent models or database records managed by your application. Laravel's authorization features provide an easy, organized way of managing these types of authorization checks.

Laravel provides two primary ways of authorizing actions: [gates](#) and [policies](#). Think of gates and policies like routes and controllers. Gates provide a simple, closure-based approach to authorization while policies, like controllers, group logic around a particular model or resource. In this documentation, we'll explore gates first and then examine policies.

Selain menyediakan layanan otentikasi bawaan, Laravel juga menyediakan cara sederhana untuk mengotorisasi tindakan pengguna terhadap sumber daya tertentu. Misalnya, meskipun pengguna diautentikasi, mereka mungkin tidak berwenang untuk memperbarui atau menghapus model Eloquent atau rekaman database tertentu yang dikelola oleh aplikasi Anda. Fitur otorisasi Laravel menyediakan cara yang mudah dan terorganisir untuk mengelola jenis pemeriksaan otorisasi ini.

Laravel menyediakan dua cara utama untuk mengotorisasi tindakan: gerbang dan kebijakan. Bayangkan gerbang dan kebijakan seperti rute dan pengontrol. Gates memberikan pendekatan otorisasi berbasis penutupan yang sederhana, sementara kebijakan, seperti pengontrol, mengelompokkan logika di sekitar model atau sumber daya tertentu. Dalam dokumentasi ini, kita akan menjelajahi gerbang terlebih dahulu, lalu memeriksa kebijakan.

Yang lebih simple itu **GATES**

* Jadi kita itu nanti ngasih PAGAR gitu yaa.. ke tempat-tempat yang user itu tidak boleh ngakses.

Dan **POLICIES** itu lebih **kompleks** dari gates.

Yang akan kita coba yaitu **GATES**

* Kalau misalnya kedepannya kita menggunakan **STARTER KIT** punya nya Laravel, seperti **JETSTREAM** atau menggunakan **BREEZE** yang di dalamnya sudah tertanam fitur autentikasi dan autorisasi yang dapat kita gunakan. Sebetulnya di belakang layar yang mereka gunakan tetep aja **GATES** dan **POLICIES**

Jadi **PENTING BAGI KITA** untuk mengetahui bagaimana laravel bekerja dengan **Authorization** ini. Sehingga nanti kedepannya ketika kita pakai STARTER KIT-nya itu udah ngerti kalau menggunakan gates/policies.

Gimana cara pakainya? Nanti kita lihat...

Tapi sekarang, kita mau coba sesuatu yang **lebih simple** dulu.

Fitur Pengelolaan Kategori

Jadi ceritanya, kita mau bikin dulu **fitur pengelolaan kategori** didalam dashboard-nya

Pada aplikasi kita yang sekarang masih baru bisa mengelola post. dan saat kita mau bikin post baru, ada kategori yang sudah kita bikin lewat **SEEDER**.

Jadi kalo kita mau nambah kategori baru ya ngga bisa, harus lewat seeder-nya lagi atau langsung kita suntikkan ke dalam database-nya. dan itu kurang oke yaa..

Jadi nanti ceritanya, kita mau nambahin menu kategori di sidebar-nya

Membuat Resource Controller : AdminCategoryController

Kita bikin Controller baru dengan nama **AdminCategoryController**

Kita bikin controller-nya dulu !!!

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:controller AdminCategoryController  
--model=Category --resource
```

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:controller DashboardPostController --model=Post  
--resource
```

Atau kita pake plugin Command Pallet di **vscode**, dengan cara **ctrl + shift / command + P** untuk mengeluarkan command paletnya. kemudian ketikkan

```
>>> Artisan: Make Controller
```

Kemudian kita tulis nama controller-nya apa... misal : **AdminController**

kemudian pilih **TIPE controller** yang **Resource**

(supaya gampang aja, walaupun di video ini hanya akan di contohkan bikin index-nya aja. karena kita sudah bisa bikin tambah edit dan hapus.. JADI SILAKAN LAKUKAN SENDIRI YA!)

Apakah akan terhubung dengan Model? **IYA**

Nama Modelnya? **Category**

maka akan otomatis dibuatkan controller-nya di dalam folder controller.

Yang sudah otomatis terhubung dengan Model Category

Karena **RESOURCE**, sudah otomatis ada **index()**, **create()**, **store()**, **show()**, **edit()**, **update()**, dan **destroy()**.

Setelah itu, kita akan bikin **ROUTE** baru di file **routes > web.php** untuk mengarah ke controller AdminCategory yang baru saja kita buat.

```
Route::resource('/dashboard/categories',
AdminCategoryController::class);
```

* Route **RESOURCE** sudah langsung memiliki semua route-nya.
sudah mengarah ke **index()**, **create()**, **store()**, **show()**, **edit()**, **update()**, dan **destroy()**.

Melihat Route di dalam aplikasi kita

Kalau kita mau melihat **ADA ROUTE APA SAJA** di dalam aplikasi kita
Kita bisa melihatnya dengan terminal berikut caranya :

```
$ php artisan route:list
```

Maka akan ditampilkan semua route yang ada di aplikasi kita.

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	
	GET HEAD	__clockwork		Clockwork\Support\Laravel\ClockworkController@webRedirect	web
	GET HEAD	__clockwork/app		Clockwork\Support\Laravel\ClockworkController@webIndex	
	POST	__clockwork/auth		Clockwork\Support\Laravel\ClockworkController@authenticate	
	PUT	__clockwork/{id}		Clockwork\Support\Laravel\ClockworkController@updateData	
	GET HEAD	__clockwork/{id}/extended		Clockwork\Support\Laravel\ClockworkController@getExtendedData	
	GET HEAD	__clockwork/{id}/{direction?}/{count?}		Clockwork\Support\Laravel\ClockworkController@getData	
	GET HEAD	__clockwork/{path?}		Clockwork\Support\Laravel\ClockworkController@webAsset	
	GET HEAD	about		Closure	
	GET HEAD	api/user		Closure	
	GET HEAD	categories			web
	GET HEAD	clockwork			api
	GET HEAD	clockwork/app			auth:api
	GET HEAD	clockwork/{path?}			web
	GET HEAD	dashboard			auth
	GET HEAD	dashboard/categories	categories.index	App\Http\Controllers\AdminCategoryController@index	web
	POST	dashboard/categories	categories.store	App\Http\Controllers\AdminCategoryController@store	web
	GET HEAD	dashboard/categories/create	categories.create	App\Http\Controllers\AdminCategoryController@create	web
	GET HEAD	dashboard/categories/{category}	categories.show	App\Http\Controllers\AdminCategoryController@show	web
	PUT PATCH	dashboard/categories/{category}	categories.update	App\Http\Controllers\AdminCategoryController@update	web
	DELETE	dashboard/categories/{category}	categories.destroy	App\Http\Controllers\AdminCategoryController@destroy	web
	GET HEAD	dashboard/categories/{category}/edit	categories.edit	App\Http\Controllers\AdminCategoryController@edit	web
	GET HEAD	dashboard/posts	posts.index	App\Http\Controllers\DashboardPostController@index	web
	POST	dashboard/posts	posts.store	App\Http\Controllers\DashboardPostController@store	auth
	GET HEAD	dashboard/posts/checkSlug		App\Http\Controllers\DashboardPostController@checkSlug	auth
	GET HEAD	dashboard/posts/create	posts.create	App\Http\Controllers\DashboardPostController@create	web
	GET HEAD	dashboard/posts/{post}	posts.show	App\Http\Controllers\DashboardPostController@show	web

Untuk route ke AdminCategory sudah ada semua.

GET HEAD	dashboard/categories	categories.index
POST	dashboard/categories	categories.store
GET HEAD	dashboard/categories/create	categories.create
GET HEAD	dashboard/categories/{category}	categories.show
PUT PATCH	dashboard/categories/{category}	categories.update
DELETE	dashboard/categories/{category}	categories.destroy
GET HEAD	dashboard/categories/{category}/edit	categories.edit
GET HEAD	dashboard/posts	posts.index

Untuk **SHOW** (menampilkan detail kategori) sepertinya kita tidak butuh.

| GET|HEAD | dashboard/categories/{category} | categories.show |

Kalau kita ngga mau ada **SHOW**, ya kita tinggal kasih tau Si Resource-nya untuk ngasih pengecualian untuk Si Show-nya.

Dengan cara menambahkan `->except('show')`

```
Route::resource('/dashboard/categories',
AdminCategoryController::class) ->except('show');
```

- * Jadi SHOW ngga akan bisa di akses, karena memang ngga di pake.
- * Jadi kalau ngga di pake, jangan dibikin supaya bisa diakses lewat URL-nya, Meskipun nanti tampilnya kosong, **JANGAN YA DEK YAA!!!**

Kalau kita cek ROUTE LIST-nya lagi, maka sekarang SHOW-nya ilang.

Nah, sekarang kita sudah punya ROUTE-nya untuk pengelolaan kategori

Sekarang kita balik lagi ke Controller nya

http > Controllers > AdminCategoryController.php

Di bagian index() kita kasih coba return untuk testing.

```
public function index()
{
    // untuk testing.
    return 'ini adalah halaman categories';
}
```

Kemudian kita masuk ke web-nya <http://127.0.0.1:8000/dashboard/categories> dan sudah bisa di akses.

Dan halaman ini bisa diakses meskipun kita LOGOUT (usernya belum login). Hal ini tidak boleh!!

Kita bisa kasih autentikasi lewat MIDDLEWARE juga, walaupun nanti akan kita ubah. untuk sementara agar tidak bisa diakses siapapun yg belum login.

Menambahkan list admin di sidebar dan Membuat Halaman Index Categories

Sekarang ceritanya, kita akan menambahkan LINK di Sidebar yang akan mengarah ke category.

Kita buka lagi Sidebar kita yang ada di Dashboard/layout, menggunakan **CTRL + P** **resource > views > dashboard > layouts > sidebar.blade.php**

Ceritanya, kita akan nambahin LIST baru, tapi kita tidak akan menambahkan
Kenapa? karena itu akan menjadi bagian yang terpisah.
Kedepannya kita bikin agar dia hanya bisa diakses oleh ADMIN.

Sebelum kita tambahkan lagi, kita tambahkan judul yang kita samakan dari template example bootstrap.

Di bagian index() kita kasih coba return untuk testing.

```
<ul class="nav flex-column">
    <li class="nav-item">
        <a class="nav-link {{ Request::is('dashboard/categories*') ? 'active' : '' }}" href="/dashboard/categories">
            <span data-feather="grid"></span>
            Posts Categories
        </a>
    </li>
</ul>
```

Kemudian kita kembali lagi ke Controller nya

http > Controllers > AdminCategoryController.php

Di bagian index() kita arahkan ke view-nya

```
public function index()
{
    // untuk testing.
    return 'ini adalah halaman categories';
}
```

Kemudian kita buat folder dan file

resource > views > dashboard > categories > index.blade.php

Kita copy-kan isinya dari index.blade.php yang **posts**

Kemudian saat kita coba cek halaman tersebut di web, maka akan terdapat error
karena katanya tidak tau variable \$posts.

karena belum kita kirimin di controller-nya.

Nah karena yang kita butuhkan data kategori, maka yang kirimkan data Category dari model

```
public function index()
{
    return view('dashboard.categories.index', [
        'categories' => Category::all()
    ]);
}
```

Kemudian kita edit satu persatu dari atas.

resource > views > dashboard > categories > index.blade.php

Dan sampai akhirnya halaman categories

<http://127.0.0.1:8000/dashboard/categories>

sudah dapat di akses.

Middleware

Menambahkan kondisi manual hanya admin/user tertentu yang bisa mengakses kategory

Nah sekarang, problemnya adalah, semua user yang login bisa masuk ke menu category.

Padahal yang kita pengen cuma user sandhika aja atau **admin** aja yang bisa lihat menu category dan bisa mengakses-nya tentu saja.

Karena aplikasi kita ngga terlalu kompleks, adminnya 1 dulu aja lah.. kedepannya kita mau nambahin **admin** ITU MUDAH !!!

Gimana caranya?

Kita coba versi yang lebih simple-nya dulu.

kita matikan dulu middleware-nya untuk route resource category.
Jadi kita ngga akan pakai middleware lagi.

Kita nanti mau ngecek manual. → Udah login atau belum. → Terus kalau udah login, dia Sandhika (Admin) atau bukan?

Halaman kategori sekarang bisa di akses oleh siapapun, ATI-ATI !!!

Tapi untuk mencegahnya, kita masuk ke Controller nya

http > Controllers > AdminCategoryController.php

Kita kasih **peng-KONDISI-an** nya di bagian index() sebelum menampilkan view-nya

Kalau semisal belum login, maka kasih pesan abort dengan status 403 (forbidden)

```
public function index()
{
    // jika belum login, maka forbidden
    if(auth()->guest()) {
        abort(403);
    }

    return view('dashboard.categories.index', [
        'categories' => Category::all()
    ]);
}
```

Kemudian kita coba akses halaman categories

<http://127.0.0.1:8000/dashboard/categories>

*tidak kembali ke login memang, tapi sekarang FORBIDDEN

Kemudian kita bikin satu kondisi lagi. Dimana jika dia bukan sandhika/admin/halofaizabd, maka kita abort juga.

```
// jika bukan halofaizabd, maka forbidden juga
```

```
if(auth()->user()->username != 'halofaizabd') {  
    abort(403);  
}
```

Kemudian kita coba akses halaman categories menggunakan akun **halofaizabd** dan akun selainnya.

<http://127.0.0.1:8000/dashboard/categories>

Tapi masalahnya, Menu “Administrator” dan list “Post Categories” masih muncul di dashboard akun yang bukan **halofaizabd**
tapi itu akan kita benerin belakangan. Yang pasti sekarang akun yang bukan **halofaizabd** udah ngga bisa masuk ke kategori.

Jadi itu simple banget, bahkan kalau misalkan kita mau nga-gabungin bisa kita jadikan 1 seperti ini

```
// jika kita gabungkan menjadi seperti ini  
if(auth()->guest() || auth()->user()->username != 'halofaizabd') {  
    abort(403);  
}
```

Kemudian kita coba akses halaman categories lagi. Pastikan Aman...

Nah bahkan kita bisa aja ngga pake **GUEST()**, ada cara lain.. itu kita bisa pakai **CHECK()**

Si **CHECK()** ini ngecek seorang user ini udah login apa belum. dan menghasilkan **TRUE** jika user sudah login.

karena kita pengen nya ngecek yang belum login, makanya didepan **AUTH()** kita tambahin **NOT**

!AUTH() -> CHECK()

Jadi jangan bingung kalau ada yang pakai **CHECK()** juga.

```
// kita juga bisa menggunakan check() dengan NOT di depannya  
if(!auth()->check() || auth()->user()->username != 'halofaizabd') {  
    abort(403);
```

```
}
```

Kelihatannya udah oke, user ngga bisa ngakses kalau dia bukan sandhika/admin/halofaizabd.

Tapi probelemnya, codingan pengkondisian tersebut harus kita **copy-copy** kan ke **semua method**.

Gimana misalkan juga kita pengen ganti, admin-nya bukan sandhika/halofaizabd.. tapi user lain.. wah harus di edit semua nya tuh. agak ribet nih.

Membuat middleware sendiri (Meng-abstraksi code tersebut)

Gimana cara meng-abstraksi code tersebut?

gimana cara mindahin codingan pengkondisian tersebut supaya bisa di pakai dengan mudah di semua tempat.

Cara paling gampang adalah kita bikin Si LOGIKA ini menjadi sebuah middleware sendiri.

KAN KEREN YAA !!!

Jadi nanti kita punya middleware sendiri yang bisa kita tambahkan di route kita dengan nama yang kita bikin sendiri.

Sehingga jika kita tambahkan middleware tersebut di route resource kategori, maka semua yang ada di route category (update,delete, dll) sudah langsung memiliki perilaku yang sama, karena dia pakai middleware di keseluruhannya.

Membuat Middleware : IsAdmin

Caranya kita bisa bikin middleware itu menggunakan Artisan

Dokumentasi Middleware :

<https://laravel.com/docs/8.x/middleware#main-content>

Kita bikin middleware baru dengan nama **IsAdmin**

Kita bikin middleware-nya dulu !!!

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:middleware IsAdmin
```

Jika ingin membuatnya dengan terminal berikut caranya :

```
$ php artisan make:middleware EnsureTokenIsValid
```

Atau kita pake plugin Command Pallet di **vscode**,
dengan cara **ctrl + shift / commad + P** untuk mengeluarkan command paletnya.
kemudian ketikkan

```
>>> Artisan: Make Middleware
```

Kemudian kita tulis nama middleware-nya apa... misal : **IsAdmin**

maka akan otomatis dibuatkan middleware-nya di dalam folder middleware.
Sebuah class yang namanya IsAdmin.

Kemudian, kita akan masukin LOGIC-nya di dalam method **HANDLE()**
dan kita nulisnya sebelum **RETURN**.

RETURN untuk menjalankan middleware berikutnya. Yang namanya middleware
itu pasti berada di antara bagian-bagian lain.

Sekarang, logic yang sudah kita buat di **AdminController.php** , kita
pindahkan (CUT) ke method **HANDLE()** di **Middleware > IsAdmin**

```
public function handle(Request $request, Closure $next)
{
    if(auth()->guest() || auth()->user()->username !== 'halofaizabd') {
```

```
        abort(403);
    }
    return $next($request);
}
```

Sekarang kalau misalkan kita jalankan/tambahkan di route resource kategori kita, belum bisa jalan karena kita belum tau nama middleware-nya apa.
Kita masih baru punya class middleware-nya saja.

Gimana caranya ngejalanin middleware kita?

Caranya yaitu kita harus masukin juga ke dalam Kernel-nya

Masuk ke file yang namanya Kernel.php

app > Http > Kernel.php

Supaya middleware kita di daftarin didalamnya.

di dalam **Kernel.php** kita punya yang namanya GLOBAL MIDDLEWARE (middleware yang otomatis jalan ketika laravel-nya jalan).

dan kita juga punya * **The application's route middleware**. (Middleware ini bisa kita kasih ke group atau ke route individualnya, satu-satu).

Sudah ada middleware yang bisa kita pakai, kecuali middleware baru kita.

Jadi kita harus daftarin dulu di sini.

```
protected $routeMiddleware = [
    ...
    // middleware kita sendiri
    'admin' => \App\Http\Middleware\IsAdmin::class,
];
```

Baru setelah itu kita bisa tambahin middleware-nya ->**middleware('admin')** di route resource category (**routes > web.php**)

```
Route::resource('/dashboard/categories',
AdminCategoryController::class) ->except('show') ->middleware('admin');
```

Kemudian kita cek di web/aplikasi kita, pastikan sudah sesuai logic-nya. Dimana hanya admin/halofaizabd yang bisa mengakses halaman kategori.

Menghilangkan list menu admin post categories pada sidebar

Sekarang yang akan kita lakukan adalah menghilangkan list menu admin post categories pada sidebar kalau dia bukan admin/halofaizabd

Caranya disini, baru kita membutuhkan fitur otorisasi menggunakan **GATE**. Karena **GATE** itu lebih flexible daripada kita menggunakan middleware. Karena kalau middleware kan gampang sebenarnya yaa, kita tinggal tambahkan middlewarenya di route yang mau kita pasang.

Tapi yang kita pengen itu sekarang bukan semuanya (semua halaman), tapi hanya **sebagian kecil** aja dari view kita.

Kita akan kasih kondisi di bagian mana yang hanya bisa tampil saat usernya adalah admin/halofaizabd

nah pakai middleware ngga bisa se-flexible itu.

Gimana cara bikin GATE???

Bagaimana cara menulis sebuah **GATE** (atau kita kasih PAGER) untuk bagian dari aplikasi kita.

Dokumentasi GATES :

<https://laravel.com/docs/8.x/authorization#writing-gates>

***GATES** adalah cara yang bagus untuk mempelajari dasar-dasar fitur otorisasi Laravel; namun, saat membangun aplikasi Laravel yang sudah cukup **kompleks**, Anda harus mempertimbangkan penggunaan **POLICIES** untuk mengatur aturan otorisasi Anda.

Gates are simply closures that determine if a user is authorized to perform a given action. Typically, gates are defined within the boot method of the App\Providers\AuthServiceProvider class using the Gate facade. Gates always receive a user instance as their first argument and may optionally receive additional arguments such as a relevant Eloquent model.

In this example, we'll define a gate to determine if a user can update a given App\Models\Post model. The gate will accomplish this by comparing the user's id against the user_id of the user that created the post:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function (User $user, Post $post) {
        return $user->id === $post->user_id;
    });
}
```

Langsung kita cobain, balik lagi ke Codingan nya
sekarang kita buka **AppServiceProvider** kita
app > Providers > AppServiceProvider.php
Di method **BOOT()** kita tambahkan dibawah **Paginator::useBootstrap();** yang sebelumnya pernah kita tambahkan

```
use App\Models\User;
use Illuminate\Support\Facades\Gate;

public function boot()
{
    Paginator::useBootstrap();

    Gate::define('admin', function(User $user) {
        return $user->username === 'halofaizabd';
    });
}
```

```
}
```

Kemudian kita buat folder dan file

resource > views > dashboard > categories > index.blade.php

Nah sekarang, kita jadi punya cara lain untuk melakukan otorisasi.

Otorisasi yang pertama, itu pakai **MIDDLEWARE** (lebih simple untuk banyak method sekaligus)

Kit juga bisa menerapkan **GATE** ke Controller di setiap method seperti sebelumnya.

http > Controllers > AdminCategoryController.php

Seperti contoh sebelumnya, Kita kasih **GATE/Pagar** di bagian index() sebelum menampilkan view-nya

```
public function index()
{
    // Jika menggunakan GATE
    $this->authorize('admin');

    return view('dashboard.categories.index', [
        'categories' => Category::all()
    ]);
}
```

Kemudian kita coba akses halaman categories

<http://127.0.0.1:8000/dashboard/categories>

*bedanya sekarang pesan error-nya **403 | This action is unauthorized.**

Sekarang, kelebihan **GATE** itu bisa dipakai dimanapun.

Seperti yang akan kita lakukan adalah **menghilangkan list menu admin post categories pada sidebar** kalau dia bukan admin/halofaizabd

```
 {{-- baris ini hanya bisa di akses oleh yang punya akses
GATE/Gerbang admin --}}
@can('admin')

<h6>
    <span>Administrator</span>
</h6>
<ul class="nav flex-column">
    <li class="nav-item">...</li>
</ul>

@endcan
```

Jika kita ingin membatasi/memberi pagar di **BLADE**, kita tidak bisa pakai **MIDDLEWARE**.

Maka dari itu kita menggunakan **GATE**

Sekarang saat kita akses dashboard, maka **pada sidebar list menu admin post categories** tidak akan muncul kalau dia bukan admin/halofaizabd

Menambahkan admin

Nah sekarang, problem barunya adalah, gimana kalau ADMIN-nya nambah. semisal kita menambahkan 1 user lagi sebagai admin.

Sebenarnya kita bisa tambahkan logic lagi di dalam middleware admin dan juga GATE yang sudah kita buat. Cuman kalau adminnya nambah atau berubah kita REPOT harus ngedit-ngedit lagi.

Menambahkan field baru di tabel user (is_admin) tanpa migrate:fresh

Nah gimana kalau misalnya, kita akan rubah rancangan aplikasi kita supaya nanti Laravelnya ngecek autorisasi lewat FIELD di dalam TABEL User kita.

Misalnya nanti kita akan nambahin 1 FIELD BARU yang namanya IS_ADMIN.

Kita bisa nambahinnya lewat MIGRASI, tapi kan **sayang banget yaaa...** migrasi yang udah kita bikin lumayan banyak, daripada kita ubah-ubah.

Daripada kita MIGRATE FRESH lagi yang menyebabkan data kita hilang semua. mending kita sisipkan saja 1 field baru tapi lewat migrasi tambahan.

Kita bisa membuat migrasi tambahan dengan terminal

Example :

```
$ php artisan make:migration add_sudah_dibaca_to_comments_table  
--table=comments
```

Jalankan

```
$ php artisan make:migration add_is_admin_to_users_table  
--table=users
```

* **flag “--table”** artinya nama tabel yang akan di modifikasi.

Atau kita pake plugin, bisa membuat migrasi dengan perintah di **vscode**, dengan cara **ctrl + shift / commad + P** untuk mengeluarkan command palette-nya. kemudian ketikkan

```
>>> Artisan: Make Migration
```

Kemudian kita tulis nama migrasi-nya apa... misal karena kita mau menyisipkan : **add_is_admin_to_users_table**

Apakah migrasi ini **membuat tabel yang baru?** pilih **NO**,

Apakah migrasi ini **memodifikasi tabel yang sudah ada?** pilih **YES**,

Apa nama tabelnya? tulisakan **users**

kemudian kita tulis model-nya apa misal : **Post**

otomatis akan dibuatkan file migrasi baru di dalam folder **migrations** yang sudah terdapat method untuk UP() dan DOWN() dan sudah terkoneksi dengan tabel **users**.

Sekarang kita buka file migrasi **add_is_admin_to_users_table.php** karena sudah otomatis terkoneksi dengan tabel **users** (tabel yang ingin kita modifikasi), jadi tinggal kita sisipkan field/kolom apa yang ingin kita tambahkan. Kita tambahan di method **UP()**

```
Schema::create('categories', function (Blueprint $table) {  
  
    $table->boolean('is_admin')->default(false);  
  
    $table->timestamps();  
});
```

* kita tambahkan field yang tipe datanya boolean, yang nama field-nya “is_admin” dan nilai default-nya “false”

untuk method **DOWN()**, kita hapus field “is_admin”-nya, siapa tau butuh rollback. kita tambahkan

```
$table->dropColumn('is_admin');
```

Sekarang kita boleh jalanin migrasi-nya caranya langsung aja ketik

```
$ php artisan migrate
```

* ini akan menjalankan migrasi yang belum dijalankan saja. Jadi ngga akan mengganggu tabel yang lain. hanya akan mengedit tabel users dengan menambah kolom baru yaitu “is_admin”.

Sekarang kalau kita lihat di database, tabel users kita sudah ada kolom **is_admin**, yang nilai defaultnya FALSE (0)

* karena nanti kita akan jadikan sandhika/halofaizabd adalah admin, maka kita ubah nilai is_admin-nya menjadi 1 manual di database. (sementara)

Enaknya kalau kita nyisipin kolom seperti ini, kalau ternyata ngga jadi nambahinnya. ya kita tinggal rollback aja!!

```
$ php artisan migrate:rollback
```

Implementasi is_admin ke middleware dan gate

Sekarang gimana implementasinya?

Sekarang kita tinggal ganti aja, logic-nya ngga lagi **username === 'sandhikagalih'** , tapi kita tinggal tulis **\$user->is_admin;**

AppServiceProvider.php

```
Gate::define('admin', function(User $user) {
    // return $user->username === 'halofaizabd';
    return $user->is_admin;
});
```

*Jadi jika **is_admin** nilainya TRUE, maka gerbangnya akan terbuka.

Tapi HATI-HATI!!!

Untuk yang middleware, nilainya harus FALSE, jadi kita tambahin NOT (!)
IsAdmin.php

```
if(!auth()->check() || !auth()->user()->is_admin) {
    abort(403);
}
```

OKEEEE.... KITA COBAAA!!!

Mantapp... Sudah OKEE!!!

[Finish]

Ahad, 09 Maret 2025 (05.14)

24. Upload ke Web Hosting (Gratis) [[LAST]]

[[PENGHJUNG SERI DARI LARAVEL 8]]

000webhost

000webhost telah berhenti beroperasi sejak 1 Oktober 2024. Situs web 000webhost.com kini diarahkan ke halaman arahan di Hostinger.com. 000webhost adalah penyedia layanan web hosting gratis yang mengumumkan akan menghentikan operasinya pada 7 Juli 2024. Penutupan penuh layanannya dijadwalkan pada 14 Oktober 2024.

Di 000webhost terdapat folder **public_html** & **folder tmp**
Kita bisa meletakkan file web kita di **public_html** semua.

nah tapi, karena kita akan bikin aplikasi Laravel kita aman, agar user itu ngga bisa sembarang ngeliat isi dari aplikasi laravel kita. Nanti akan kita pisahkan yang boleh di lihat oleh public dimasukkan ke folder **public_html**, lalu sisa-nya sistem laravel kita akan kita simpan di **ROOT**-nya (/). atau di luar folder tersebut.

Jadi akan kita siapkan 2 hal.

Kita **duplicat** folder **project** kita, coba-laravel, menjadi folder “**laravel**”
Kemudian kita keluarkan folder **public**

Kemudian kita compress menjadi ZIP kedua folder tersebut “laravel” dan “public”. dan sebaiknya jangan di RAR.

Selanjutnya, sebelum kita upload ke hosting.
kita export database blog kita terlebih dahulu.

Export method nya kita pilih **CUSTOM**,
pastikan semua tabel terceklist
pastikan **save output to a file > GO**

Fitur

Yang pertama akan kita lakukan adalah, kita **upload** dulu file **laravel.zip** di halaman **root/home** (diluar folder **public_html**),

kemudian masuk ke folder **public_html**, dan kita **upload** file **public.zip** nya, dan file **.htaccess** bisa kita hapus saja.

kita coba extract di folder tersebut menggunakan fitur yang sudah disediakan oleh 000webhost, namun terjadi ERROR **ftp_put()**. Jadi kita ngga bisa nyimpan file nya di folder tersebut.

nah untuk ngakalinnya, kita butuh sebuah **library** untuk **meng-extract**.

Unzipper Github

<https://github.com/ndeet/unzipper>

kita hanya butuh file **unzipper.php** saja.

file **unzipper.php** akan kita upload ke dalam folder **public_html**, supaya nanti kita akses lewat web-nya langsung.

tugas dari file **unzipper.php** ini adalah melakukan **unzip** dari file yang ada di web hosting kita

nah, file **unzipper.php** ini tinggal kita akses di web kita.

<http://wpublog.000webhostapp.com/unzipper.php>

dengan **unzipper** ini, kita bisa melakukan **unzip** atau **membuat zip**.

Pilih file zip yang akan di extract,
extraction path jika **di kosongi**, maka akan di extract ke folder yang sama.

kemudian kita extract juga yang **laravel.zip**, kita pindahkan saja ke folder **public_html** biar gampang.

kita pindahkan semua yang ada di dalam folder **public** ke folder **public_html**, dan kita pindahkan folder **laravel** ke luar

Jadi folder **public_html**, didalamnya adalah file yang bisa diakses oleh user, jangan sampai file laravel nya bisa di akses oleh user, makanya kita pindahin ke luar.

Karena kita punya file **.env** yang terdapat username password database kita. **BAHAYA** kalau sampai bisa diakses.

HTTP ERROR 500

nah tapi sekarang kalau kita jalankan web-nya **HTTP ERROR 500**

ada yang harus kita perbaiki dulu.

karena sekarang di **public_html**, yang pasti pertama kali dijalankan adalah **index.php**

index-nya perlu kita edit, supaya dia langsung mengakses laravel-ny.

edit bagian **autoloader-nya** dan **bootstrap** untuk menjalankan aplikasi kita

```
require __DIR__.'/../laravel/vendor/autoload.php';

$app = require_once __DIR__.'/../laravel/bootstrap/app.php';
```

PHP VERSION

sekarang, kalau kita cek web kita, maka akan terjadi error katanya, dependensi dari composer kita butuh PHP versi $\geq 7.3.0$

yang artinya si 000webhost masih menggunakan versi 7.3 harus kita ubah agar pakai versi yang lebih baru.

manage website > menu website setting > general > php version > ubah ke yang terbaru.

Sekarang aplikasi web kita sudah bisa diakses oleh siapapun

IMPORT DATABASE & KONFIGURASI .ENV

namun, halaman blog-nya error KATANYA NGGA ADA DATABASE-nya

manage website > home > menu Tools > menu database manager > new database > isi nama database, username dan password

catat password.

db_name dan db_username-nya akan di generate otomatis

import database ke phpmyadmin

konfigurasi file .env

```
APP_NAME=Faiz_Blog
#APP_ENV=local
APP_ENV=production
APP_KEY=base64:2dnDD5TWkk5uJbne/0QKP8t1Q8h4BBIFvQ6TTzPUj1E=

APP_DEBUG=true
#APP_DEBUG=false

APP_URL=http://faizwebblog.infinityfreeapp.com/

LOG_CHANNEL=stack

DB_CONNECTION=mysql
DB_HOST=sql1206.infinityfree.com
DB_PORT=3306
DB_DATABASE=if0_38478271_faiz_blog
DB_USERNAME=if0_38478271
DB_PASSWORD=F84WcCdXJEaf
```

dan sekarang database-nya sudah terhubung.

APLIKASINYA SUDAH BERJALAN.

Problem symbolic link

tapi sayangnya ada 1 fitur yang ngga jalan, yaitu fitur **UPLOAD IMAGE**.
sepertinya keterbatasan web hosting-nya karena tidak bisa melakukan **SYMBOLIC
LINK**

kalaupun kita coba bikin post baru, maka postnya berhasil ditambahkan tapi gambarnya
tidak tampil.

padahal kalaupun kita cek di **file manager**, di folder **storage > app > public >
post-image**

gambar ada (berhasil di upload)

tapi waktu kita membahas mengenai upload file kita belajar yang namanya
SYMBOLIC LINK, gimana caranya menghubungkan **folder storage** kita ke folder
public

```
$ php artisan storage:link
```

sayangnya, untuk web hosting gratis ini tidak memberikan kita akses ke terminal
atau ke SSH (fitur berbayar).

IMPORT

Sebetulnya ada cara untuk ngakalinnya.
kita bisa panggil **\$php artisan storage:link** lewat codingan kita. sayangnya fitur itu
tidak jalan.

misalnya kita mau coba,
caranya adalah kita bikin aja sebuah **route baru**

di web.php

```
Route::get('/storage-link', function() {
    Artisan::call('storage:link');
});
```

* kalau kita arahkan ke url **/storage-link** , akan menjalankan sebuah perintah.

tapi ternyata error, katanya ngga bisa manggil sebuah **method** yang namanya **symlink()**

karena dinonaktifkan dengan alasan keamanan.

[Finish]

Kamis, 09 Maret 2025 (00.00)

Tutorial Upload Aplikasi Laravel ke Hosting Melalui cPanel

<https://www.jagoanhosting.com/tutorial/tips-hosting/upload-aplikasi-laravel-hosting>

Infinityfree
rumahweb
heroku
danil

<http://faizwebblog.infinityfreeapp.com/>

[[LAST]]

* Upload .zip ke infinityfree menggunakan FileZilla tidak bisa lebih dari 10 MB

Folder dengan Size Besar di Laravel

① **vendor/** (10MB - 200MB)

💡 Kenapa besar?

- Berisi semua package yang diinstal via **Composer**
- Bisa mencapai **ratusan MB** jika banyak package digunakan

✓ Solusi Mengurangi Size:

- Pastikan hanya menginstal package yang dibutuhkan

Jangan upload **vendor/ langsung**, tapi jalankan perintah ini di server setelah upload:

bash

CopyEdit

```
composer install --no-dev --optimize-autoloader
```

• Penjelasan:

- **--no-dev** → Menghapus package development (seperti testing & debug tools)
- **--optimize-autoloader** → Memperkecil ukuran autoloader untuk meningkatkan performa

Update Terakhir

[Update terakhir, video ke - 24 menit ke 00:00 / 00:00]

https://youtube.com/playlist?list=PLFIM0718LjIWiihbBIq-SWP0U6b6x21Q_2&si=lzJu8C3YTHLEOGA

```
 href="/dashboard/posts/{{ $post->slug }}/edit"
```

DashboardPostController

```
public function edit(Post $post)
{
    // halaman buat nampolin ubah data
    return view('dashboard.posts.edit', [
        'post' => $post,
        'categories' => Category::all()
    ]);
}
```

resources > views > dashboard > posts > show.blade.php

Rangkuman

Instalasi dan Running Project

Instalasi

```
$ composer create-project laravel/laravel=8.0 nama-aplikasi  
--ignore-platform-reqs
```

Running project

```
$ php artisan serve
```

Kalo misalkan kita mau **bekerja dengan data**, itu kita harus simpan codingan-nya **di dalam MODEL**.

Dan kalau misalkan **ada proses** seperti memilih dan menampilkan view, itu biasanya dilakukan **di dalam controller**.

Koneksi Database

- jika kita masuk ke folder config/database.php akan ada konfigurasi database

```
'default' => env('DB_CONNECTION', 'mysql'),
```

konfigurasi defaultnya dia manggil fungsi yg namanya env() parameter pertamanya '**DB_CONNECTION**' parameter keduanya '**mysql**'

Tinggal kita update file **.env**

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

kita migrate fresh lagi

```
$ php artisan migrate:fresh --seed
```

Membuat menu active pada navbar/sidebar dengan request

Caranya yaitu

```
{{ Request::is('dashboard') ? 'active' : '' }}
```

```
<li class="nav-item">
    <a class="nav-link {{ Request::is('dashboard/posts') ? 'active' : '' }}" href="/dashboard/posts">
        <span data-feather="file-text"></span>
        My Posts
    </a>
</li>
```

Membuat Model Post

Setelah membuat model, kita bikin migrationnya. untuk membuat skema tabel post kita. jadi kita bikin 2 kali.

Tapi ada caranya agar bisa membuat model dan migrasinya sekaligus satu kali command.

Jika ingin membuatnya dengan terminal berikut caranya :

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

nanti akan dibuatkan 2 buah file. yang pertama modelnya, kedua migrasinya dengan nama tabel yang sudah plural (jamak) = Posts.

Sedangkan nama model kita itu singular = Post

```
Model created successfully.  
Created Migration: 2024_04_13_143059_create_posts_table
```

Jika kita kembali ke codingan kita, maka di dalam folder **model** kita sudah ada file **Post.php** dan udah include semua yang kita butuhkan.

app > Models > Post.php

dan juga yang penting pada folder **database > migrations** sekarang nambah 1 file lagi, yaitu **2024_04_13_143059_create_posts_table.php**

Life Hack / Shortcut

- * duplikat baris di vscode dengan **Alt + Shift + Bawah**
- * Mengganti kata/variabel yang sama sekaligus di vscode dengan **Ctrl + D** (4x atau sampai kata/variabel yang sama yang akan diganti)
- * Kalau setelah update CSS, dan halaman tidak berubah. Kita **refresh chace-nya (Ctrl + Shift + R)**
- * Untuk me-navigasi antar file di dalam laravel menggunakan vscode, kalau mau cepet dan kita sudah tau nama file-nya apa. (bahkan kita tidak perlu buka buka lagi sidebar navigator.
Kita cukup pencet **Ctrl/Command + p > ketik nama file yang ingin dibuka** (semisal web.php)
- * Kita buka dulu **DashboardPostController**
Kita cari **method store()**
Dari pada scroll-scroll, kita bisa **Ctrl + P > tulis @store**

Additional

(Additional)

Menghapus 1 baris data di MySQL workbench

Klik kanan baris yang akan dihapus > **Delete Row (s) > Apply**

Mengetahui versi laravel

```
$ php artisan --version
```

Mengupdate composer

```
$ composer update
```

kalaupun terjadi error saat kita mau update composer setelah clone project laravel orang lain, agar menyesuaikan dengan versi PHP kita

```
$ composer update --ignore-platform-reqs
```

Untuk menambahkan keterangan saat data-nya kosong, kita bisa menggunakan `@forelse`

```
@forelse ($posts as $post)

<tr>
    <td>{{ $loop->iteration }}</td>
    <td>{{ $post->title }}</td>
    <td>{{ $post->category->name }}</td>
    <td>
        <a href="/dashboard/posts/{{ $post->slug }}" class="badge bg-info"><span data-feather="eye"></span></a>
        <a href="" class="badge bg-warning"><span data-feather="edit"></span></a>
        <a href="" class="badge bg-danger"><span data-feather="x-circle"></span></a>
    </td>
</tr>

@empty
<tr>
    <td colspan="4" class="text-center">Data tidak tersedia</td>
</tr>

@endforelse
```

Other Reference

<https://medium.com/@jpmakangiras/local-web-development-menggunakan-laravel-valet-24979d9ceb95>

Menyimpan File/Gambar Publik dan Pribadi di Laravel

https://laraveldaily.com.translate.goog/post/how-to-store-public-private-files-images-laravel?_x_tr_sl=en&_x_tr_tl=id&_x_tr_hl=id&_x_tr_pto=tc

Mengetahui Versi Laravel

<https://fahmialazhar.com/cara-cek-versi-laravel-menggunakan-command-prompt-terminal>

Fixed Error (additional)

- **open_basedir restriction in effect. File(/) is not within the allowed path(s):**
- **laravel error : Namespace declaration statement has to be the very first statement or after any declare call in the script [closed]**
- **Laravel 9: Undefined type 'Illuminate\Support\Facades\Route'. Intelephense 1009**

<https://stackoverflow.com/questions/29318709/how-can-i-resolve-your-requirements-could-not-be-resolved-to-an-installable-set>

Solved : Your requirements could not be resolved to an installable set of packages
<https://www.youtube.com/watch?v=c0ngIxGop8I>

Install Clockwork jika tidak berhasil

```
$ composer require itsgoingd/clockwork
```

Tambahkan

```
$ composer install --ignore-platform-reqs
```

atau

```
$ composer update --ignore-platform-reqs
```

I don't care if they steal my idea. I care that they don't have any of their own.

~nikola tesla

Saya tidak peduli jika mereka mencuri ide saya. Saya peduli jika mereka tidak memiliki ide mereka sendiri. ~nikola tesla