LARAVEL 10 & BOOTSTRAP 5.3

QUICK LARAVEL

CEPAT DAN MUDAH MEMBUAT BLOG DENGAN LARAVEL

TONI LISTIYO

Table of Contents

Pendahuluan **9**

Alasan Belajar Laravel 10

Disclaimer 11

Grup dan Source Code 12

Lingkungan Kerja **13**

Kebutuhan Sistem 14

Lain - Lain 15

Instalasi **16**

Install Project 17

Templating 18

Autentikasi **23**

Install Laravel UI 24

Konfigurasi database 25

Fitur Kategori **26**

Model dan Migration 27

Controller 29

View index 30

Konfigurasi Route 31

Form Tambah Kategori 32

Menyimpan Kategori ke Database 35

Menampilkan Daftar Kategori 36

Menampilkan Flash Message 39

Update Kategori 41

Hapus Kategori 44

Validasi **46**

Menambahkan Validasi 47

Menampilkan Pesan Error 48

Middleware **49**

Fitur Post **50**

Model dan Migration 51

Enum 52

Controller 54

Konfigurasi Route 55

View Index 56

Form Tambah Post 58

Menyimpan Post ke Database 60

Menampilkan Daftar Post 62

Update Post 64

Hapus Post 67

Membuat Navbar Link 68

One to Many Relationship **69**

Konfigurasi 70

Menampilkan Relasi 74

N+1 Query Problem 76

Upload Gambar **81**

Konfigurasi Filesystem 82

Upload Cover Post 84

Menampilkan Gambar 86

Update Gambar 88
Hapus Gambar 90

Pagination **91**
Membuat Factory 92

Fitur Setting **95**
Model dan Migration 96
Route 97
View 98
Controller 99
Konfigurasi Upload Banner 101
Create and Update Setting 102
Setting Model 104

Homepage **106**
Title Bar 108
Banner 109
Navbar 110
Fix Error Setelah Login 112

Halaman Post **114**
Post di Homepage 115
Exceprt Post 117
Single Post 119
Post Berdasarkan Kategori 122

Publish Post **124**
Badge Status 128
Save and Publish Post 130

Upgrade Laravel 10 dan Bootstrap 5.3 ..... **135**
Upgrade ke Laravel 10 136
Upgrade ke Bootstrap 5.3 137

Dark Mode **138**
Implementasi Dark Mode 139
Switch Dark Mode 141

Ekstrak Form menjadi Component **150**
Ekstrak Input Text 151
Ekstrak Textarea 154
Ekstrak Select 155
Ekstrak Input File 158
Ekstrak Card 159
Ekstrak Layout Content 160
Ekstrak Button 162

Refactoring **163**
Blade Service Injection 164
Model Events 168
Form Request Validation 172

Authorization **180**
Membuat Gate 181
Otorisasi Aksi Menggunakan Gate 183
Policy 184
Otorisasi Aksi Menggunakan Policy 187
Otorisasi Melalui Blade 189

Role and Permission **191**
Install Library 192
Membuat Role 193
Mendapatkan User Berdasarkan Role 197
Mendapatkan Role User 198
Mengecek Apakah User Memiliki Role Tertentu 199
Membatasi Akses Menggunakan Role 200
Menampilkan Role di User Profil 203

Notifikasi Email **204**
Konfigurasi Mail Server 205
Verifikasi User via Email 208
Queue 212

Mailable Class **215**
Invite User 216
Membuat InvitationMail 223
Content Email 226
Modifikasi Register 229
Mailable vs Notification 232

Komentar **233**
Membuat Model, Migration dan Controller 234
Relasi Model 236
Policy 238
Controller 239
Route 240
View 241
Notifikasi 247

Penutup **254**

Section Bonus 255

Deploy Ke VPS 256

Pendahuluan

Selamat datang di ebook "**Quick Laravel : Cepat dan Mudah Membuat Blog dengan Laravel**"! Blogging telah menjadi cara yang populer untuk berbagi pemikiran, pengalaman, dan informasi dengan audiens online. Dalam ebook ini, Anda akan belajar cara membuat blog dengan cepat dan mudah menggunakan framework PHP populer, yaitu Laravel.

Laravel adalah salah satu framework PHP yang paling populer dan mudah digunakan. Dalam ebook ini, kami akan mengajarkan cara memanfaatkan Laravel untuk membuat blog dengan cepat dan mudah. Kami akan membahas langkah-langkah dasar untuk membuat blog, termasuk mengatur database, membuat model, view, dan controller.

Ebook ini dirancang untuk pemula dalam pengembangan web yang ingin mempelajari cara membuat blog menggunakan Laravel dengan cepat dan mudah. Anda akan belajar bagaimana menggunakan Laravel untuk membuat aplikasi web yang dinamis dan dapat diakses dari berbagai perangkat.

Setelah menyelesaikan ebook ini, Anda akan memiliki pengetahuan dasar dan keterampilan untuk membuat blog dengan Laravel. Kami berharap ebook ini membantu Anda memulai perjalanan Anda dalam dunia pengembangan web dan meningkatkan kemampuan Anda dalam membuat aplikasi web yang efektif. Selamat membaca dan selamat belajar!

Alasan Belajar Laravel

Berikut ini adalah beberapa alasan mengapa kita perlu belajar Laravel:

1. Sintaks Ekpresif

   Laravel memiliki sintaks yang ekspresif dan jelas, sehingga mudah dipahami oleh pemula maupun profesional. Sintaks ini membantu mempertahankan konsistensi dalam proyek dan mempermudah pemahaman kode oleh orang lain. Contohnya, dengan melihat `User::all()`, orang akan memahami bahwa kode tersebut digunakan untuk mendapatkan semua user.

2. Rapid Development

   Laravel mengikuti desain Model-View-Controller (MVC) yang membantu memisahkan logika aplikasi dari antarmuka pengguna. Ini membuat aplikasi lebih mudah dikelola dan di-debug, dan sintaks yang jelas mempercepat pengembangan. Laravel menghasilkan basis kode yang bersih dan dapat dipelihara dan aplikasi yang dapat diskalakan.

3. Framework PHP Terpopuler

   Laravel adalah framework PHP terpopuler di Github dengan 71.900 bintang, jauh lebih populer dibandingkan Codeigniter 4 dengan 4.500 bintang dan Symfony dengan 27.800 bintang (pada saat tulisan ini dibuat).

4. Ekosistem yang besar

   Ekosistem Laravel sangat besar, baik resmi maupun dari library atau package pihak ketiga.

5. Komunitas yang Aktif

   Laravel memiliki komunitas yang aktif di Facebook, Telegram, dan memiliki konferensi resmi bernama Laracon yang biasanya berlangsung setiap tahun. Konferensi ini biasanya menampilkan pembicara dari komunitas Laravel yang berbagi pengetahuan dan pengalaman mereka tentang Laravel dan pengembangan aplikasi web.

6. Peluang Kerja Besar

   Karena merupakan framework PHP paling populer, peluang menjadi programmer Laravel sangat bagus, baik di dalam negeri maupun luar negeri.

Disclaimer

- Isi ebook ini adalah tutorial atau modul jadi mungkin tidak akan banyak menjelaskan teori.
- Saya asumsikan pembaca ebook ini minimal sudah bisa CRUD menggunakan Laravel.
- Saya usahakan penulisan code dalam ebook ini sesuai standar laravel (https://github.com/alexeymezenin/laravel-best-practices), sehingga saya berharap ebook ini bisa menjadi referensi untuk menulis kode laravel sesuai standar.
- Dilarang menyebarkan ***full version ebook*** ini kepada siapapun. Membeli ebook ini berarti setuju

Lingkungan Kerja

Development Environment atau lingkungan pengembangan adalah sebuah lingkungan komputer yang diperlukan oleh seorang pengembang untuk menjalankan, menguji, dan memperbaiki aplikasi atau sistem. Biasanya, lingkungan ini terdiri dari web server dan database. Biasanya, pengembang harus menginstal dan mengkonfigurasi web server dan database secara manual. Namun, dengan menggunakan tools tertentu, pengembang dapat mengatasi konfigurasi web server dan database dengan mudah, tanpa perlu melakukan konfigurasi secara manual.

Laragon adalah solusi pengembangan lingkungan yang handal bagi pengguna Windows dan pastikan untuk memilih versi PHP ≥ 8.0 (versi 8.1 ke atas lebih baik). Bagi pengguna MacOs dan Linux, Valet / Valet Linux dapat menjadi pilihan yang baik untuk mengembangkan aplikasi web.

Kedua tools tersebut mempermudah proses pembuatan *Development Environment*. Laragon menyediakan segala fitur yang dibutuhkan seorang pengembang seperti PHP, Apache, dan database dalam satu paket yang terintegrasi. Sementara Valet menawarkan solusi pengembangan web yang cepat dan mudah, khususnya untuk MacOs. Valet Linux adalah versi dari Valet yang dapat dijalankan pada sistem operasi Linux. Kedua tools ini juga mempermudah proses pembuatan virtual host dan juga menangani konfigurasi yang rumit sehingga membuat pengembangan aplikasi lebih mudah dan efisien.

Kebutuhan Sistem

Laravel yang akan kita gunakan dalam project ini adalah versi 9 (ada panduan upgrade ke versi 10 untuk versi premium). Pastikan sudah memenuhi persyaratan untuk menggunakan Laravel 9, diantaranya :

- PHP ≥ 8.0
- BCMath PHP Extension
- Ctype PHP Extension
- cURL PHP Extension
- DOM PHP Extension
- Fileinfo PHP Extension
- JSON PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PCRE PHP Extension
- PDO PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Lain - Lain

Instal juga tools dibawah ini

1. Composer

Untuk menginstal library yang kita butuhkan.

2. Laravel Installer

Digunakan Untuk instal laravel

3. Node JS

Instalasi

Judul ebook ini menunjukkan bahwa kita akan membuat sebuah studi kasus tentang bagaimana membuat blog menggunakan Laravel. Dalam studi kasus ini, kita akan fokus pada pembuatan dua jenis tampilan, yaitu tampilan untuk pengunjung dan tampilan untuk admin.

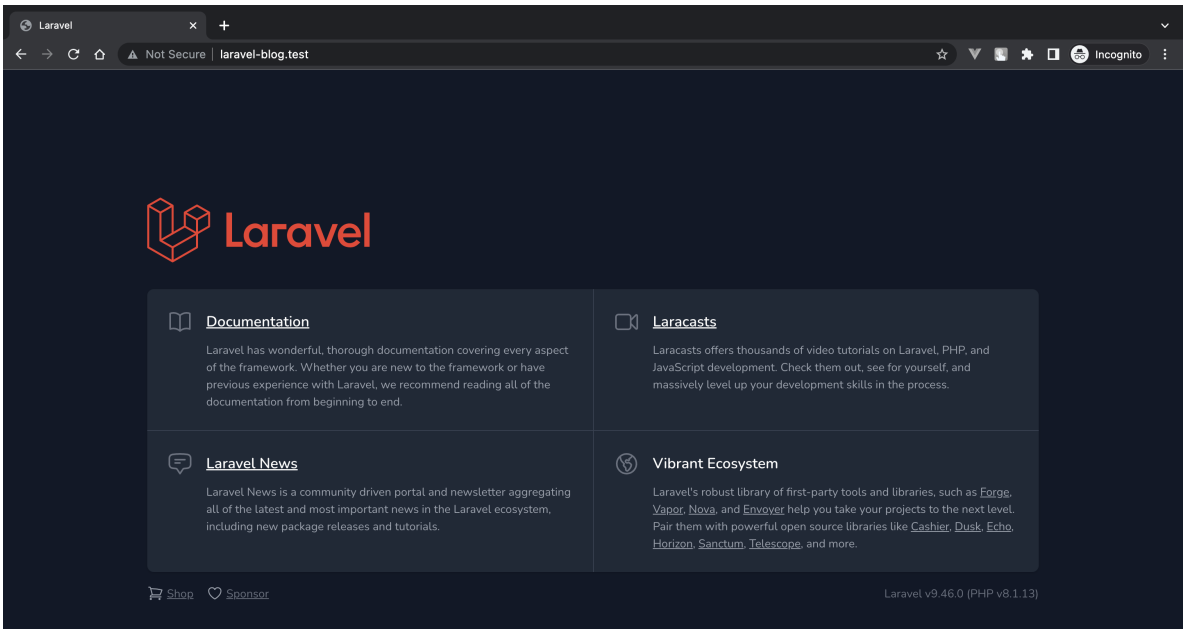
Install Project

Untuk memulai, langkah pertama yang harus dilakukan adalah menginstal project Laravel melalui perintah berikut

```
laravel new laravel-blog
```

Setelah proses instalasi selesai, langkah selanjutnya adalah masuk ke direktori proyek menggunakan perintah `cd laravel-blog` di terminal. Kemudian, untuk menjalankan aplikasi, kita perlu menggunakan perintah `php artisan serve`. Setelah itu, buka browser dan akses alamat http://localhost:8000 untuk melihat tampilan awal dari blog yang telah dibuat menggunakan Laravel.

Jika menggunakan Valet atau Laragon sebagai server lokal, Kita dapat mengakses blog yang telah dibuat menggunakan Laravel melalui alamat http://laravel-blog.test pada browser . Pastikan Anda telah melakukan konfigurasi server dengan benar sehingga aplikasi dapat berjalan dengan lancar pada alamat tersebut.

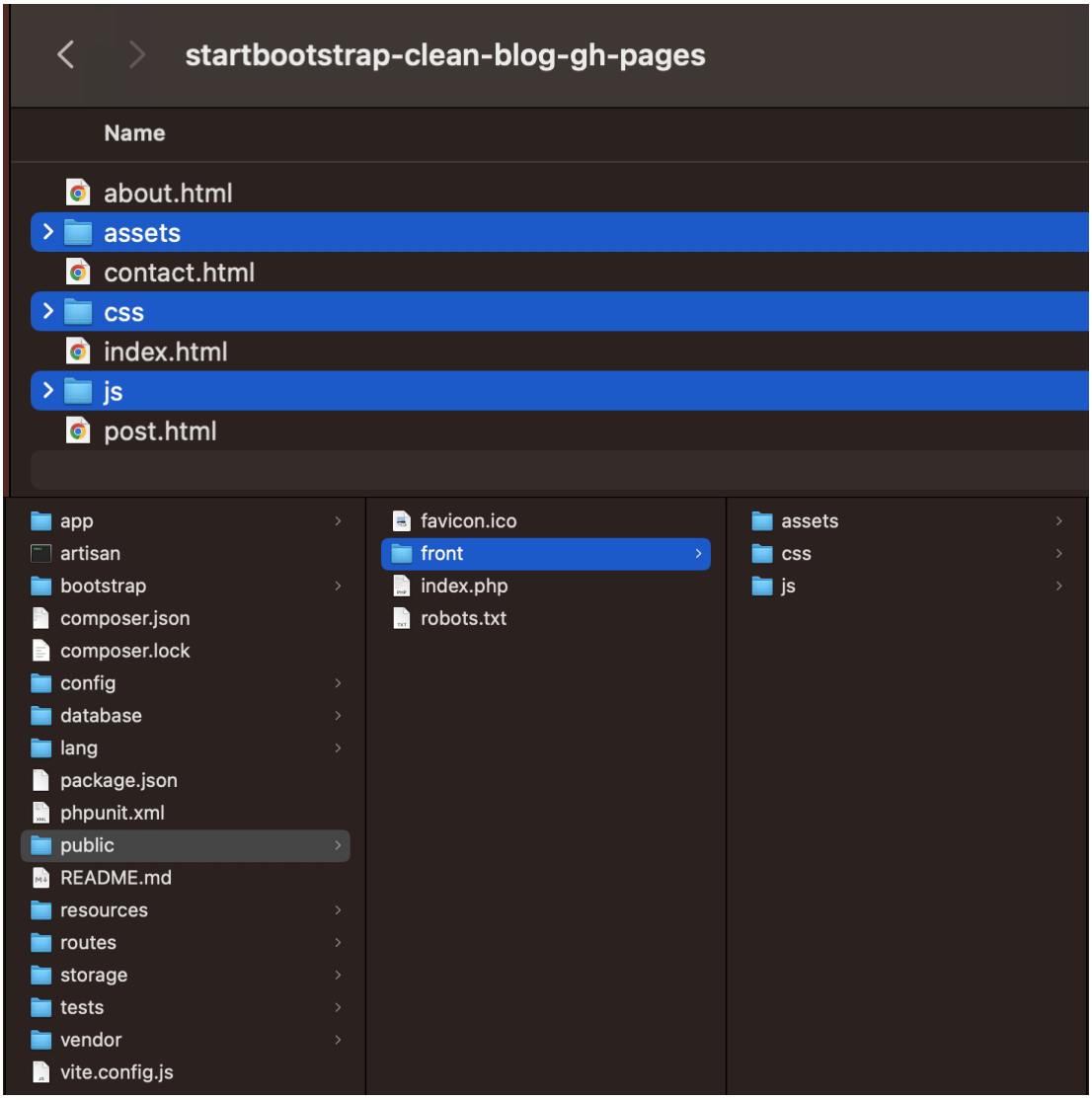


Templating

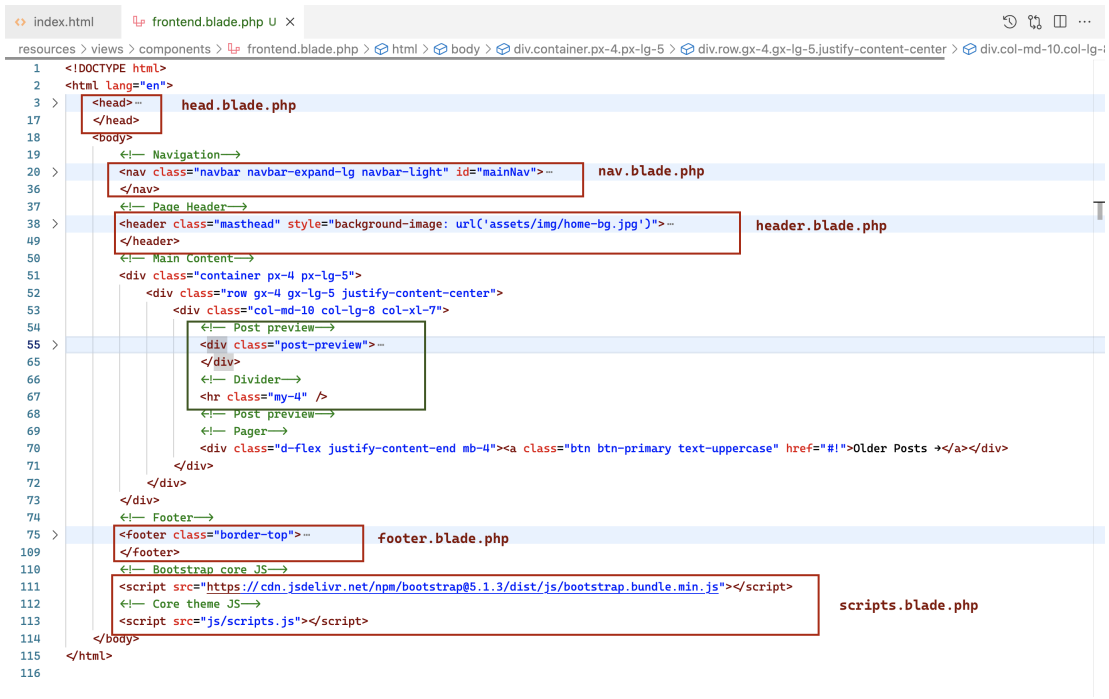
Untuk menghindari redundansi dan mempermudah pemeliharaan, kita akan menggunakan teknik templating. Ini memungkinkan kita memisahkan bagian header, footer, dan bagian lain yang tidak berubah dari sebuah halaman web dari bagian yang akan berubah. Template akan membuat tampilan blog kita terlihat menarik dan mempermudah pengelolaan. Kita akan menggunakan tema Clean Blog yang dibangun dengan Bootstrap 5.

Langkah - langkah templating sebagai berikut:

1. Pindahkan folder assets, css, dan js dari tema yang kita download ke dalam folder `public/front`.



2. Buka dan copy seluruh source code file `index.html` dan paste ke `resources/components/front/frontend.blade.php`.



3. Selanjutnya buat file - file partials (lihat gambar di atas) di dalam folder `resources/views/front` , pindahkan source code yang berada di dalam kotak merah kedalam masing - masing file partials. Sehingga isi folder front seperti ini :

```
- views
 + components
 - front
  - footer.blade.php
  - head.blade.php
  - header.blade.php
  - nav.blade.php
  - scripts.blade.php
```

4. Kita perlu menghubungkan asset yang ada di dalam file `head.blade.php` dan `scripts.blade.php` dan juga `header.blade.php` untuk menampilkan gambar

```
<!-- resources/views/front/head.blade.php -->
<head>
<!-- Kode Lain -->
<link href="{{asset('front/css/styles.css')}}" rel="stylesheet" />

</head>
```

```
<!-- resources/views/front/scripts.blade.php -->
<!-- Kode lain -->
<script src="{{asset('front/js/scripts.js')}}"></script>
@stack('scripts')
```

```
<!-- resources/views/front/header.blade.php -->
<header class="masthead" style="background-image:
url({{asset('front/assets/img/home-bg.jpg')}})">
<!-- Kode Lain -->
</header>
```

Untuk menambahkan fleksibilitas dan efisiensi, kita akan menggunakan `@stack` blade directive pada file `scripts.blade.php` .Fungsi dari `@stack` adalah untuk merender script dari halaman lain pada template yang sedang dibuat.

Ini memungkinkan kita untuk menambahkan *script* ke file blade tanpa harus mengedit file blade utama dan memastikan bahwa *script* yang ditambahkan akan ditempatkan pada posisi yang tepat dalam halaman web yang dibuat.

Helper `asset` digunakan untuk memanggil file external yang berada di dalam folder `public`.

5. Tambahkan file - file partials tadi ke dalam file `frontend.blade.php` dengan directive `@include` , hasil akhirnya seperti ini

```
<!DOCTYPE html>
<html lang="en">
  @include('front.head')
  <body>
    @include('front.nav')
    @include('front.header')
    <div class="container px-4 px-lg-5">
      <div class="row gx-4 gx-lg-5 justify-content-center">
        <div class="col-md-10 col-lg-8 col-xl-7">
          {{$slot}}
          <hr class="my-4" />
          <div class="d-flex justify-content-end mb-4">
            <a class="btn btn-primary text-uppercase" href="#!">
              Older Posts →
            </a>
          </div>
        </div>
      </div>
    </div>
    @include('front.footer')
    @include('front.scripts')
  </body>
</html>
```

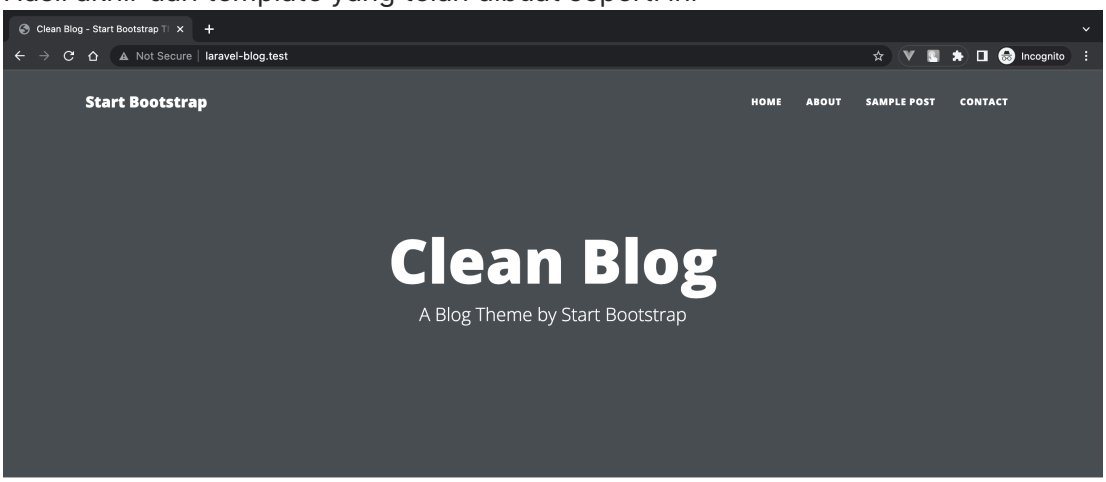
Source code yang berada di dalam kotak berwarna hijau (lihat gambar point no 2) merupakan tampilan yang nantinya dinamis. Variabel `$slot` merupakan bawaan dari laravel, variabel ini berfungsi sebagai tempat tampilan dari template child.

6. Untuk menguji template yang telah dibuat, edit file `welcome.blade.php`.

```
<x-frontend>
  <div class="post-preview">
    <a href="post.html">
      <h2 class="post-title">
        Man must explore, and this is exploration at its greatest
      </h2>
      <h3 class="post-subtitle">
        Problems look mighty small from 150 miles up
      </h3>
    </a>
    <p class="post-meta">
       Posted by
      <a href="#!">Start Bootstrap</a>
       on September 24, 2022
    </p>
  </div>
</x-frontend>
```

Variable `$slot` yang berada di dalam file frontend.blade.php akan di-replace dengan source code yang berada didalam tag x-frontend .

7. Hasil akhir dari template yang telah dibuat seperti ini



Man must explore, and this is exploration at its greatest

Problems look mighty small from 150 miles up

Posted by Start Bootstrap *on September 24, 2022*

Autentikasi

Autentikasi bertujuan untuk memastikan bahwa hanya user tertentu yang memiliki akses pada suatu sistem . Kenapa fitur ini saya buat diawal? Karena kita akan memanfaatkannya juga untuk template admin. Ada banyak cara untuk membuat autentikasi di laravel baik menggunakan library maupun manual, karena project ini menggunakan bootstrap, jadi kita pakai saja yang sesuai kebutuhan , ada tiga pilihan yaitu : manual, menggunakan Fortify dan yang ketiga menggunakan library Laravel UI . Kita akan menggunakan Laravel UI karena sudah disediakan halaman - halaman yang dibutuhkan untuk proses autentikasi.

Install Laravel UI

Laravel UI adalah *library* yang menyediakan kerangka dasar dan tampilan autentikasi dasar untuk aplikasi Laravel. Mencakup template yang sudah disediakan untuk login, registrasi, reset password, dan verifikasi email, serta route, controler, dan middleware yang diperlukan untuk fitur-fitur tersebut. Jadi kita tidak perlu membuat secara manual. *Library* ini dirancang untuk dapat bekerja dengan berbagai *framework* front-end seperti Bootstrap dan Vue.js.

Untuk menginstal Laravel UI jalankan perintah dibawah ini:

```
composer require laravel/ui
```

Setelah selesai instal, jalankan perintah untuk menghasilkan kerangka kerja front-end Bootstrap dengan fungsi autentikasi.

```
php artisan ui bootstrap --auth
```

Setelah install laravel ui dan juga *framework frontend*-nya, kita perlu untuk menginstal dan mengompilasi(compile) dependensi *frontend* dengan perintah :

```
npm install && npm run dev
```

Jika kita menuju halaman http://laravel-blog.test/register atau http://laravel-blog.test/login , akan muncul halaman register dan login.

> **Warning:** Pastikan npm run dev tetap aktif

Konfigurasi database

Setelah menginstal Laravel UI dengan Bootstrap, Kita harus mengkonfigurasi database agar aplikasi dapat terhubung ke database dan menyimpan data. Konfigurasi database berada di file `.env`. Di project ini kita akan menggunakan database MySQL dengan nama database `laravel_blog`, parameter `DB_USERNAME` dan `DB_PASSWORD` sesuaikan dengan *credential* masing - masing. Seperti ini konfigurasinya.

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

Setelah Kita mengkonfigurasi file `.env`, Laravel akan mengambil konfigurasi database dari file ini. Jangan lupa untuk membuat database dengan nama `laravel_blog` kemudian untuk menguji koneksi database ,jalankan perintah `php artisan migrate`. Jika koneksi berhasil, Laravel akan membuat tabel-tabel database yang telah dibuat di file migration.

Silahkan mencoba register user baru !

Fitur Kategori

Dalam bab ini, kita akan membahas bagaimana cara membuat fitur kategori blog pada aplikasi yang kita buat. Kategori blog memungkinkan kita untuk mengelompokkan postingan kita berdasarkan tema tertentu seperti teknologi, bisnis, atau hobi.

Dalam proses membuat fitur ini, kita akan menggunakan migrasi untuk membuat tabel kategori dan model untuk memanipulasi data kategori. Kita juga akan membuat halaman yang menampilkan daftar kategori dan form untuk menambah atau mengubah kategori.

Setelah itu, kita akan mengintegrasikan fitur kategori pada fitur postingan sehingga kita dapat memasukkan postingan ke dalam kategori tertentu. Dengan membuat fitur ini, kita akan lebih memahami cara kerja Laravel dan mengasah kemampuan kita dalam membangun aplikasi web.

Model dan Migration

Mari kita mulai dengan membuat model sekaligus migrationnya. Model digunakan untuk merepresentasikan tabel di database, sedangkan migration digunakan untuk mengelola dan memperbarui struktur tabel didatabase.

Best Practice dalam membuat model adalah sebisa mungkin untuk menggunakan Bahasa Inggris dan gunakan *singular case* / kata tunggal. Jadi untuk fitur kategori kita akan membuat model dengan nama `Category` dengan menjalankan perintah berikut :

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

Dengan menambahkan flag `-m` , laravel akan secara otomatis membuatkan file migrationnya. Buka file migration dan ubah menjadi seperti ini

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

Jalankan perintah `php artisan migrate` untuk menggenerate tabel `categories` di database.

Pada file `Category` tambahkan property `$guarded`. Property `guarded` dalam model Laravel adalah properti yang digunakan untuk menentukan kolom-kolom pada tabel database yang tidak diizinkan untuk diisi secara massal menggunakan method `create` atau `fill`. Kolom-kolom tersebut akan dianggap sebagai guarded atau terlindungi.

Secara default, Laravel memiliki properti `guarded` yang kosong, yang berarti bahwa semua kolom pada tabel database dapat diisi secara massal menggunakan method `create` atau `fill`. Karena kolom - kolom pada tabel `categories` tidak menyimpan data sensitif, kita buat properti `guarded` kosong.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
  use HasFactory;
  protected $guarded = [];
}
```

Controller

Untuk membedakan halaman administrator dan pengunjung, kita akan membuat controller untuk administrator di dalam folder khusus bernama "**Admin**". Ini akan memastikan bahwa akses ke halaman administrator hanya dapat dilakukan oleh user yang memiliki akses yang sesuai dan mempermudah pemeliharaan aplikasi.

Buat controller untuk category dengan perintah :

```
php artisan make:controller Admin/CategoryController -r
```

Flag `-r` digunakan untuk membuat controller baru dengan fungsi *resource* yang sudah ditentukan. Fungsi *resource* adalah fungsi yang digunakan untuk mengelola data di dalam aplikasi (CRUD) yang terdiri dari fungsi : `index`, `create`, `store`, `show`, `edit`, `update`, `destroy`.

Untuk saat ini cukup buat controller dulu, logika pemrogramman kita bahas nanti.

View index

Untuk membedakan tampilan antara halaman admin dan pengunjung, kita akan memisahkan view-nya. View untuk admin akan ditempatkan di dalam folder "**admin**" dan di dalamnya kita akan membuat folder "**categories**". Semua tampilan yang berkaitan dengan kategori akan ditempatkan di dalam folder ini.

Buat file `index.blade.php` di dalam folder `categories`, isi dengan kode berikut :

```
@extends('layouts.app')
@section('content')
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="card">
        <div class="card-header">Categories</div>
        <div class="card-body">
        </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

Perhatikanlah bagaimana penerapan layout untuk halaman admin berbeda dengan halaman pengunjung. Halaman admin menggunakan *Template Inheritance* dengan *child view*-nya `categories/index.blade.php` yang menginduk dari view `layouts/app.blade.php` yang merupakan default dari library `laravel/ui`.

Directive `@section` memiliki fungsi yang sama dengan variabel `$slot` pada template halaman untuk pengunjung. Selanjutnya, kita akan memanggil view ini pada `CategoryController` melalui method `index` .

```
class CategoryController extends Controller
{
  public function index()
  {
    return view('admin.categories.index');
  }
}
```

Konfigurasi Route

Untuk mengatur jalur akses fitur kategori, kita perlu menambahkan route baru di dalam file `web.php` yang berada di dalam folder routes. Route ini akan mengarahkan permintaan pengguna ke `CategoryController` dengan menentukan method yang akan dipanggil sesuai dengan tipe permintaan (GET, POST, dll) yang diterima.

```
<?php
use App\Http\Controllers\Admin\CategoryController;

Route::group(['prefix'=>'admin'],function(){
  Route::resource('categories',CategoryController::class);
});
```

Dengan menggunakan `Route::resource`, Laravel akan secara otomatis membuatkan route yang terkait dengan operasi CRUD pada method resource controller, sehingga mempermudah kita dalam membuat route tanpa harus menuliskannya satu per satu secara manual.

Jika ingin mengetahui route apa saja yang dibuat dengan method `resource` dapat menjalankan perintah `php artisan route:list`, outputnya seperti ini :

```
GET       admin/categories                  # menampilkan daftar category
POST      admin/categories                  # form untuk menambahkan category
GET       admin/categories/create           # menyimpan category baru ke
database
GET       admin/categories/{category}       # menampilkan category tertentu
PUT|PATCH admin/categories/{category}       # mengupdate category yang telah
diedit
DELETE    admin/categories/{category}       # menghapus category
GET       admin/categories/{category}/edit # form untuk mengedit category
```

Untuk mengecek apakah konfigurasi route sudah benar, buka http://laravel-blog.test/admin/categories dibrowser kalian, maka akan muncul halaman index kategori.

Form Tambah Kategori

Form tambah kategori digunakan untuk menambah kategori baru ke database yang berasal dari input user melalui sebuah form. Selain itu, kita juga akan membahas tentang validasi dan handling form submit untuk memastikan data yang diterima dalam form benar dan aman.

Buat file `categories/create.blade.php`, kodenya seperti ini:

```
@extends('layouts.app')
@section('content')
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="card">
        <div class="card-header">Tambah Kategori</div>
        <div class="card-body">
          <form method="POST" action="{{route('categories.store')}}">
            @csrf
            <div class="mb-3">
              <label for="name" class="form-label">Category</label>
              <input type="text" class="form-control" id="name" name="name">
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

`@csrf` adalah *directive* Blade pada Laravel yang berfungsi untuk menambahkan token *Cross-Site Request Forgery (CSRF)* pada form HTML. Token ini digunakan untuk memastikan bahwa permintaan yang diterima oleh aplikasi adalah dari sumber yang sah dan dapat dipercaya, dan menghindari serangan *CSRF*.

Laravel secara otomatis menambahkan token ini pada setiap form yang dibuat dengan menggunakan method `POST` , `PUT` , atau `DELETE` sehingga kita hanya perlu menambahkan `@csrf` pada form untuk memastikan aplikasi kita aman dari serangan *CSRF*.

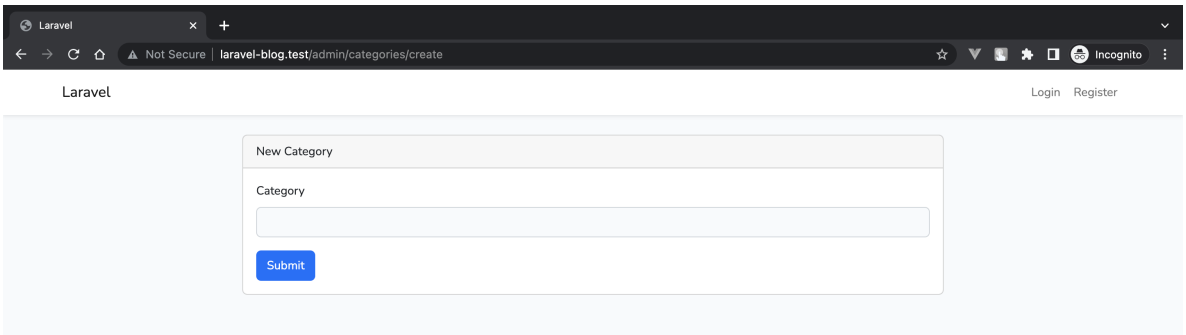
Action pada form diatas menggunakan *route name*. *Route name* memudahkan kita dalam melakukan redirect atau membuat link menuju route tertentu tanpa perlu mengetahui URL lengkap. Selain itu, *route name* juga mempermudah dalam mengubah URL tanpa perlu memperbarui setiap bagian dari aplikasi yang mengarah ke URL tersebut. Dengan

route name, kita bisa memastikan bahwa aplikasi tetap berfungsi dengan baik meskipun URL berubah.

Langkah selanjutnya adalah menghubungkan form tambah kategori ke *controller* agar dapat diakses melalui browser.

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class CategoryController extends Controller
{
  // Source Kode lain
  public function create()
  {
    return view('admin.categories.create');
  }
  // Source kode lain
}
```

Untuk mengujinya, buka http://laravel-blog.test/admin/categories/create di browser. Jika langkah - langkah diatas diikuti dengan benar akan muncul form untuk menambahkan kategori baru.



Menambahkan Tombol Tambah Kategori

Tombol tambah kategori berguna untuk mengarahkan menuju form tambah kategori ketika diklik, ubah view `index.blade.php` menjadi seperti ini :

```
<div class="col-md-8">
  <div class="d-flex flex-row  mb-3">
    <a href="{{route('categories.create')}}" class="btn btn-primary">Tambah
Kategori</a>
  </div>
  <div class="card">
    <div class="card-header">Kategori</div>
    <div class="card-body">
      <table class="table table-bordered">
        {{-- Kode lainnya --}}
      </table>
    </div>
  </div>
</div>
```

Menyimpan Kategori ke Database

Langkah selanjutnya adalah menyimpan kategori yang diinput melalui form ke dalam database. Proses insert data kategori ini dilakukan melalui function `store` pada controller `CategoryController`

```
public function store(Request $request)
{
  Category::create([
    'name'=>$request->input('name'),
    'slug'=>Str::slug($request->input('name'))
  ]);
  return redirect()->route('categories.index');
}
```

`Illuminate\Http\Request` adalah Class di Laravel yang bertanggung jawab untuk memproses permintaan HTTP yang masuk ke aplikasi. Class ini menyediakan berbagai method yang mempermudah kita dalam menangani informasi yang terkait dengan permintaan, seperti method input yang bisa digunakan untuk memperoleh data dari form yang dikirimkan.

Method `create` pada model `Category` bertujuan untuk mempermudah proses insert data baru ke dalam database. Method ini menerima sebuah array yang berisi data yang akan disimpan, lalu menambahkan data tersebut ke dalam tabel yang terkait dengan model yang bersangkutan. Dengan menggunakan metode `create` , kita bisa menyimpan data dengan cepat dan efisien tanpa harus menulis query insert secara manual.

Helper `Illuminate\Support\Str::slug()` berfungsi untuk mengubah string ke dalam format URL yang *SEO friendly*. Dalam konteks ini, "slug" mengacu pada bagian URL yang terkait dengan judul. Sebagai contoh, jika judul blog adalah "Cara Membuat Blog dengan Laravel", slug yang dihasilkan adalah "cara-membuat-blog-dengan-laravel".

Setelah data berhasil disimpan ke dalam database, controller akan mengarahkan ke halaman index category dengan menggunakan helper `redirect` . Ini akan memastikan bahwa pengguna dapat melihat data kategori yang baru saja ditambahkan dan memastikan bahwa proses insert data telah selesai dengan sukses.

Menampilkan Daftar Kategori

Setelah data kategori berhasil disimpan ke dalam database, kita perlu untuk menampilkan daftar kategori tersebut di aplikasi yang kita buat. Daftar kategori akan ditampilkan melalui method `index` pada `CategoryController`. Method ini akan memanggil data dari database melalui model `Category` dan menampilkan hasilnya dalam bentuk tabel atau daftar yang bisa dilihat oleh pengguna.

```
public function index()
{
  $categories = Category::all();
  return view('admin.categories.index',compact('categories'));
}
```

> **Warning:** Selalu gunakan plural untuk penamaan variabel yang berisi lebih dari satu data, gunakan nama `$categories` daripada `$category` atau `$dataCategory` dan biasakan menggunakan Bahasa Inggris. Ini merupakan kode yang disarankan.

Kita mengambil semua data kategori yang ada di database dengan menggunakan method `all()` pada model `Category` . Hasil dari method ini selanjutnya diteruskan atau dikirim ke view `index.blade.php` melalui controller `CategoryController`.

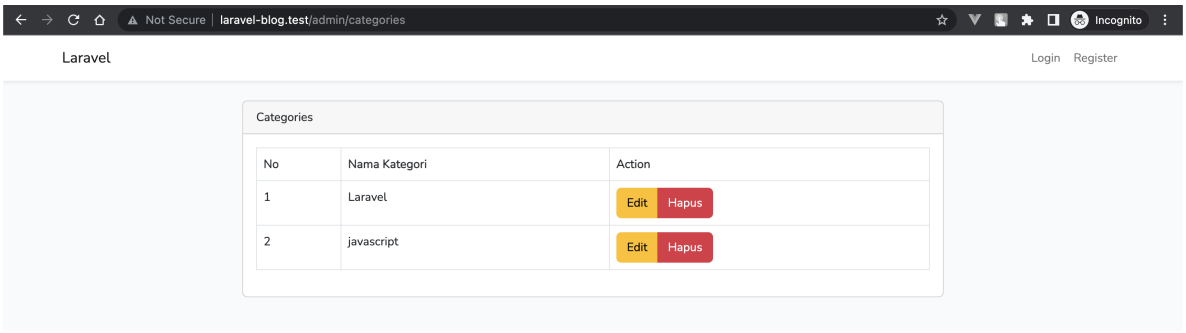
Pada file `index.blade.php` data kategori yang sudah diterima dari controller akan diteruskan lagi menggunakan directive `@foreach` untuk ditampilkan sebagai daftar kategori dalam bentuk tabel.

```
<div class="card-body">
 <table class="table table-bordered">
  <thead>
   <tr>
    <th>No</th>
    <th>Nama Kategori</th>
    <th>Action</th>
   </tr>
  </thead>
  <tbody>
   @foreach ($categories as $category)
   <tr>
    <td>{{$loop->iteration}}</td>
    <td>{{$category->name}}</td>
    <td>
     <div class="btn-group">
      <a href="#" class="btn btn-warning">Edit</a>
      <a href="#" class="btn btn-danger">Hapus</a>
     </div>
    </td>
   </tr>
   @endforeach
  </tbody>
 </table>
</div>
```

`$loop->iteration` adalah property dari object `$loop` yang disediakan oleh Blade. Property ini digunakan untuk mengetahui urutan perulangan saat melakukan looping menggunakan directive `@foreach` .

Misalnya, jika melakukan looping pada sebuah data array, `$loop->iteration` akan menyimpan angka dimulai dari 1 yang menunjukkan urutan dari setiap item dalam array tersebut. Ini berguna jika kita ingin menampilkan nomor urut pada tabel atau mengambil data dari array dengan urutan tertentu.

Pada halaman index kategori sudah muncul daftar kategori. Untuk sementara aksi untuk tombol edit dan hapus kita kosongkan dulu.


Not Secure | laravel-blog.test/admin/categories
Incognito
Laravel
Login
Register
Categories
No
Nama Kategori
Action
1
Laravel
Edit
Hapus
2
javascript
Edit
Hapus

Menampilkan *Flash Message*

Flash message adalah pesan singkat yang muncul untuk memberi tahu pengguna tentang status operasi yang terjadi, seperti kesalahan validasi, sukses menyimpan data, atau kegagalan operasi lainnya. *Flash message* ini ditampilkan hanya untuk satu kali, biasanya setelah operasi berhasil atau gagal dilakukan.

Untuk membuat *flash message* pertama buat file baru `layouts/alert.blade.php`

```
@if (session('success'))
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <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>
    </div>
  </div>
</div>
@endif
```

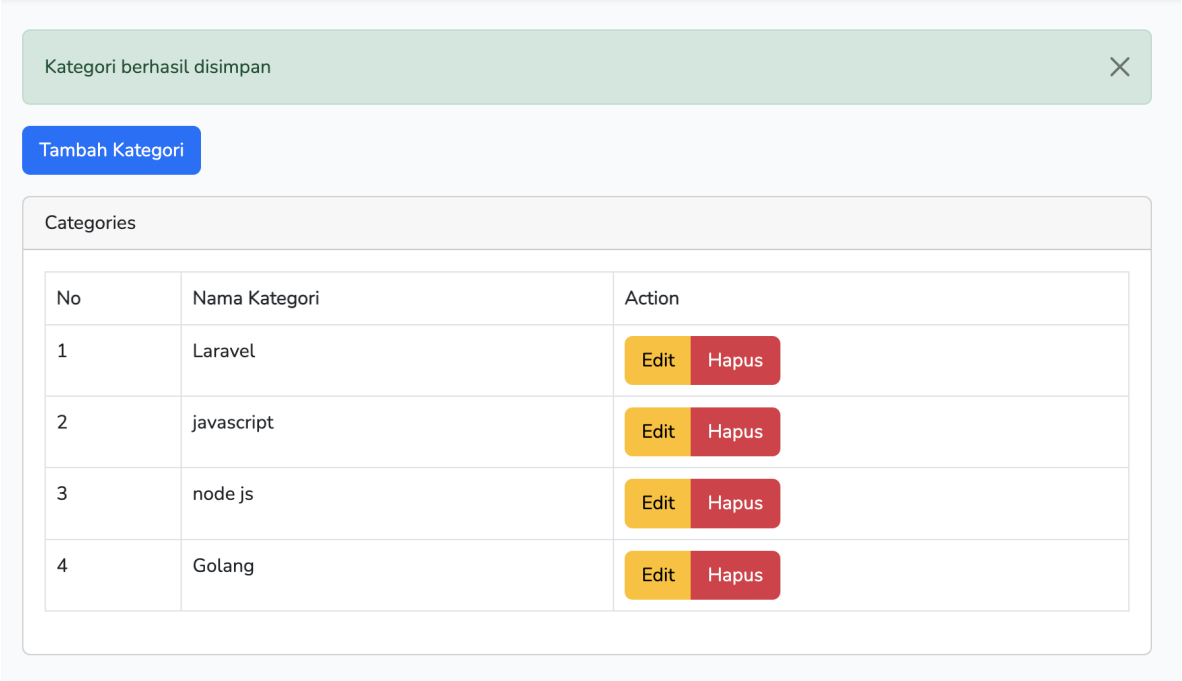
Sisipkan / include `alert.blade.php` ke dalam `layouts/app.blade.php`

```
<main class="py-4">
  @include('layouts.alert')
  @yield('content')
</main>
```

Pada `Controller` di method `store` tambahkan method with saat `redirect` ke halaman index.

```
public function store(Request $request)
{
  Category::create([
    'name'=>$request->input('name'),
    'slug'=>Str::slug($request->input('name'))
  ]);

  return redirect()->route('categories.index')->with('success','Kategori
berhasil disimpan');
}
```

Ketika kategori baru berhasil disimpan, akan muncul pesan berhasil. Itu tandanya *flash message* berhasil dibuat.



Update Kategori

Salah satu fitur yang sering dibutuhkan dalam aplikasi web adalah fitur update. Fitur ini memungkinkan kita untuk mengubah data yang sudah tersimpan di database. Dengan fitur ini, kita tidak perlu menghapus data yang sudah ada dan menambahkan data yang baru, cukup dengan mengubah data yang sudah ada saja.

Pertama update tombol aksi edit yang berada di index kategori agar menuju ke halaman edit kategoru ketika diklik.

```
<a href="{{route('categories.edit',$category)}}" class="btn btn-
warning">Edit</a>
```

Pada `CategoryController` di method edit ubah paramater `$id` menjadi model binding. Model binding adalah cara Laravel untuk mengambil data dari database dan menampilkannya di halaman web. Model binding memudahkan dalam mengelola data, karena tidak perlu menuliskan query untuk mengambil data dari database.

```
//dengan model binding
public function edit(Category $category)
{
  return view('admin.categories.edit',compact('category'));
}

// Tanpa model binding
public function edit($id)
{
  $category = Category::find($id);
  return view('admin.categories.edit',compact('category'));
}
```

Kemudian buat file view dengan nama `edit.blade.php` untuk menampilkan halaman edit.

```
@extends('layouts.app')
@section('content')
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="card">
        <div class="card-header">Edit Kategori</div>

        <div class="card-body">
         <form method="POST"
action="{{route('categories.update',$category)}}">
            @csrf
            @method('PUT')
            <div class="mb-3">
              <label for="name" class="form-label">Category</label>
              <input type="text" value="{{$category->name}}" class="form-
control" id="name" name="name">
            </div>
            <button type="submit" class="btn btn-primary">Update</button>
          </form>
        </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

Pada form edit diatas terdapat blade directive `@method('PUT')`, perlu kalian ketahui bahwa browser hanya mendukung method `POST` dan `GET`, saya yakin perbedaan keduanya kalian sudah paham. Karena HTTP verb untuk route `categories.update` adalah `PUT` diperlukan *method spoofing* dengan menambahkan directive `@method('PUT')`.

Dengan menambahkan directive `@method('PUT')` pada elemen form, Laravel akan menambahkan atribut `_method` dengan nilai `"PUT"` pada elemen form HTML yang dihasilkan. Ini akan memungkinkan kita mengirimkan permintaan PUT ke route yang sesuai melalui form.

Pada method update di controller, kita gunakan lagi model binding, sehingga kodenya akan seperti ini

```
public function update(Request $request, Category $category)
{
  $category->update([
      'name'=>$request->input('name'),
      'slug' => Str::slug($request->input('name'))
  ]);
  return redirect()->route('categories.index')->with('success', 'Kategori
berhasil diupdate');
}
```

Notice: HTTP verb seperti *PUT*, *PATCH*, dan *DELETE* tidak bisa dipanggil secara langsung melalui browser karena browser tidak memiliki fitur untuk mengirimkan permintaan HTTP dengan verb tersebut. Oleh karena itu, biasanya HTTP verb ini hanya bisa dipanggil melalui form HTML.

Hapus Kategori

Untuk menghapus kategori kita tidak bisa langsung menggunakan url `categories/1` . Kita perlu membuat form dengan menambahkan method spoofing `DELETE` agar tombol hapus berfungsi. Ubah tombol aksi menjadi seperti di bawah ini, pada tombol hapus ubah menjadi sebuah form

```
<div class="d-flex flex-row">
<a href="{{route('categories.edit',$category)}}" class="btn btn-
warning">Edit</a>
 <form action="{{ route('categories.destroy', $category) }}" method="POST">
   @csrf
   @method('DELETE')
   <button class="btn btn-danger" type="submit">Hapus</button>
 </form>
</div>
```

Pada controller method `destroy` tambahkan perintah untuk menghapus kategori yang dipilih.

```
public function destroy(Category $category)
{
 $category->delete();
 return redirect()->route('categories.index')->with('success', 'Kategori
berhasil dihapus');
}
```

Atau tanpa menggunakan model binding kita bisa menggunakan method `destroy`

```
public function destroy($id)
{
 Category::destroy($id);
 return redirect()->route('categories.index')->with('success', 'Kategori
berhasil dihapus');
}
```

Menambahkan Konfirmasi Hapus

Fitur hapus yang sudah dibuat akan langsung menghapus kategori saat klik tombol hapus. Namun, untuk mencegah terhapus secara tidak sengaja, perlu ditambahkan konfirmasi sebelum menghapus.Ini bisa dilakukan dengan menambahkan event `onclick`

pada tombol hapus yang akan menampilkan konfirmasi sebelum melakukan aksi hapus.

```
<button onclick="return confirm('Apakah anda yakin akan hapus?')" class="btn
btn-danger" type="submit">Hapus</button>
```

Validasi

Validasi form berfungsi untuk memvalidasi input yang diberikan oleh pengguna sebelum data tersebut dikirim ke server atau digunakan dalam aplikasi. Ini bertujuan untuk memastikan bahwa data yang diterima sesuai dengan format yang diharapkan dan memenuhi persyaratan yang ditentukan. Ini dapat meliputi pemeriksaan seperti memastikan bahwa suatu field tidak kosong, memvalidasi format email, atau memastikan bahwa suatu nomor telepon benar.

Untuk menambahkan validasi dapat menggunakan method validate yang disediakan oleh object `Illuminate\Http\Request` , jika validasi lolos kode selanjutnya akan dieksekusi, jika gagal akan mengembalikan exception `Illuminate\Validation\ValidationException` .

Menambahkan Validasi

Sebelum kategori disimpan ke database, perlu dilakukan validasi apakah input user sudah sesuai dengan format yang ditentukan. Di database kolom name di tabel `categories` tidak boleh kosong, oleh karena itu perlu membuat validasi untu memastikan bahwa user telah mengisi field nama kategori di form kategori sebelum proses insert ke database. Selain itu nama kategori juga harus unik. Agar lebih memahami, mari tambahkan validasi di `CategoryController`

```
public function store(Request $request)
{
  $request->validate([
    'name'=>'required|unique:categories,name'
  ]);
}
```

Format validasi di atas artinya field name di form wajib diisi dan harus unik, selengkapnya tentang rule validasi dapat dilihat di Dokumentasi.

Menampilkan Pesan Error

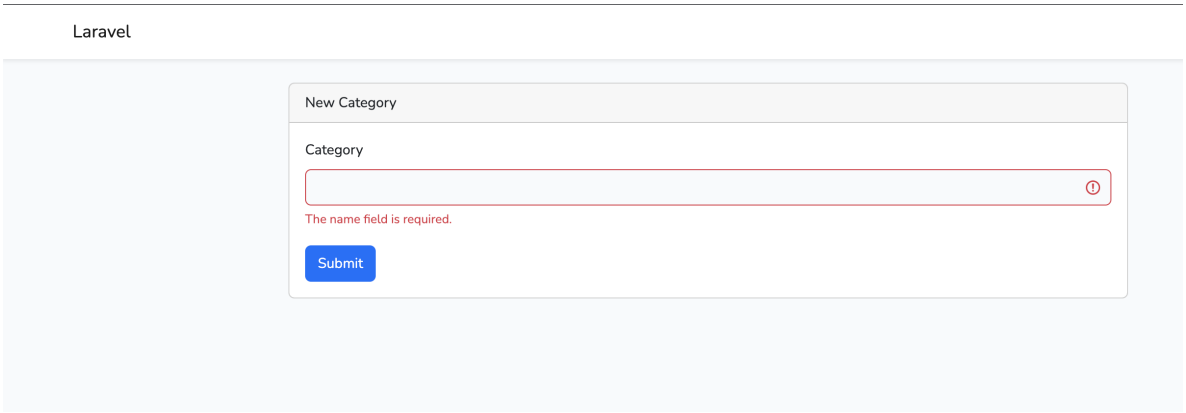
Setelah validasi selesai dibuat,penting untuk menampilkan pesan kesalahan agar pengguna tahu ada kesalahan atau kegagalan dalam proses validasi yang dilakukan. Dengan adanya pesan error, pengguna dapat dengan mudah mengidentifikasi masalah yang terjadi dan mencari solusi untuk mengatasinya yaitu dengan mengisi field sesuai dengan aturan validasi.

Untuk menampilkan pesan error di form tambah kategori, seperti ini kodenya

```
<div class="mb-3">
  <label for="name" class="form-label">Category</label>
  <input type="text" class="form-control @error('name') is-invalid @enderror"
id="name" name="name">
  @error('name')
    <div class="invalid-feedback">
      {{$message}}
    </div>
  @enderror
</div>
```

`@error('name') is-invalid @enderror` artinya jika terdapat `error` pada input field bernama '`name`', maka class '`is-invalid`' akan diterapkan pada elemen input field.

Ketika mencoba klik submit tanpa input nama kategori maka akan muncul pesan error.



Middleware

Middleware merupakan mekanisme untuk memeriksa dan menyaring permintaan HTTP yang masuk ke aplikasi. Sebagai contoh, Laravel menyediakan middleware untuk memverifikasi apakah pengguna telah diautentikasi atau belum. Jika pengguna belum diautentikasi, middleware akan mengarahkan pengguna ke halaman login. Namun, jika pengguna telah diautentikasi, middleware akan memperbolehkan permintaan untuk melanjutkan ke dalam aplikasi.

Halaman kategori admin dapat diakses tanpa harus login terlebih dahulu. Ini karena belum ada middleware di aplikasi. Tambahkan `auth` midleware di file `web/routes.php`, kodenya seperti berikut :

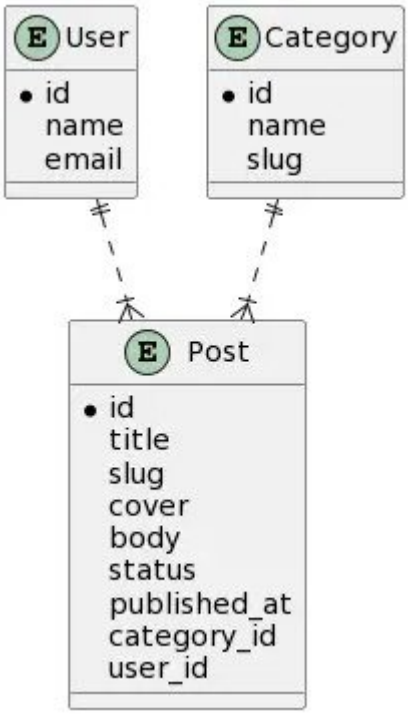
```
Route::group(['prefix'=>'admin','middleware'=>'auth'],function(){
  Route::resource('categories',CategoryController::class);
});
```

Dengan menambahkan midleware `auth` , jika mengakses halaman kategori admin tanpa login, aplikasi akan mengarah ke halaman login.

Fitur Post

Fitur post ini digunakan untuk menambahkan, mengedit dan menghapus post. Fitur ini juga dilengkapi dengan validasi dan otomasi untuk memastikan bahwa post yang ditambahkan sesuai dengan aturan yang ditentukan. Kedepannya, fitur post juga akan dilengkapi dengan fitur untuk mengelola komentar (ada di ebook full version).

Adapun, antara kategori dan post memiliki relasi *one-to-many*, yang berarti bahwa satu kategori dapat memiliki banyak post, sementara post hanya dimiliki oleh satu kategori. Di bawah ini adalah ERD untuk Category, Post dan User.



Model dan Migration

Penjelasan mengenai model dan migration sudah dijelaskan di bab `fitur kategori` jadi mari langsung saja kita buat model beserta migrationnya dengan perintah :

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

Struktur untuk tabel `posts` sebagi berikut :

```
public function up()
{
  Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('slug');
    $table->string('cover')->nullable()->comment('image cover');
    $table->text('body');
    $table->string('status');
    $table->date('published_at')->nullable();
    $table->foreignIdFor(Category::class)->constrained();
    $table->foreignIdFor(User::class)->constrained();
    $table->timestamps();
  });
}
```

`foreignIdFor` adalah method di Laravel yang digunakan untuk menentukan *foreign key* pada sebuah tabel.

Method `foreignIdFor` memiliki dua parameter yaitu `$model` dan `$column`. Parameter `$model` berisi nama model referensi, sedangkan parameter `$column` berisi nama kolom referensi yang akan dijadikan sebagai *foreign key*.

Jika parameter ke dua tidak diisi, method ini akan mengambil *primary key* dari model referensi dan akan membuat kolom *foreign key* dengan format `$model_{primary_key}` pada tabel saat tabel tersebut di-create.

Sebagai contoh `$table->foreignIdFor(Category::class)` akan menghasilkan kolom `category_id` di tabel `posts`. Jika *primary key* di model `Category` adalah `id_category` maka akan menghasilkan kolom `category_id_category` di tabel `posts`.

Dengan menggunakan metode `constrained()` pada setiap kolom, maka *foreign key* akan diatur secara otomatis untuk merujuk ke *primary key* pada masing-masing tabel referensi.

Enum

Mungkin kalian bertanya kenapa kolom status menggunakan string? kenapa tidak enum? Ini pendapat pribadi saya, menggunakan string sebagai tipe data untuk kolom status adalah pilihan yang lebih fleksibel daripada menggunakan enum.

String memberikan kemampuan untuk menambahkan status baru tanpa harus mengubah definisi tipe data pada file migration. Misalnya, jika kita ingin menambahkan status `archived`, kita dapat melakukannya dengan menambahkan string baru tanpa perlu melakukan perubahan pada file migration.

Namun, menggunakan enum memiliki keuntungan tersendiri seperti membatasi jenis status yang valid dan membuat aplikasi lebih terstruktur.

Solusi agar jenis status tetap konsisten meskipun tipe kolom string yaitu dengan melakukan konversi attribut status di model `Post` kedalam enum.

Untuk mengonversi atribut model ke dalam enum, kita perlu membuat sebuah enum terlebih dahulu, kita akan membuat enum di folder `app\Enums`, buat terlebih dahulu folder `Enums`.

```
<?php

namespace App\Enums;

enum StatusEnum:string
{
  case DRAFT = 'draft';
  case PUBLISHED = 'published';
  case ARCHIVED = 'archived';
}
```

Setelah membuat enum, kita dapat menambahkan casting ke dalam model `Post`. Casting ini akan mengubah tipe data dari atribut status dari string menjadi enum `StatusEnum`. Berikut adalah penggunaannya:

```php
<?php

namespace App\Models;

use App\Enums\StatusEnum;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
  use HasFactory;
  protected $guarded = [];

  protected $casts = [
    'status'=>StatusEnum::class
  ];
}
```

Dengan mengonversi atribut model ke dalam enum, kita dapat memastikan bahwa nilai atribut selalu valid dan sesuai dengan daftar nilai yang telah didefinisikan dalam enum.

Controller

Buat controller baru untuk mengelola semua operasi yang terkait dengan Post dengan perintah :

```
php artisan make:controller Admin/PostController -r --model=Post
```

Option `--model` berfungsi untuk menentukan model yang akan digunakan oleh controller, dengan option `--model` , kita tidak perlu melakukan cara manual untuk *Model Binding*.

Konfigurasi Route

Setelah selesai membuat controller, tambahkan route agar method - method di dalam controller dapat merespon permintaan pengguna.

```
Route::group(['prefix'=>'admin','middleware'=>'auth'],function(){
  Route::resource('categories',CategoryController::class);
  Route::resource('posts',PostController::class);
});
```

Jika terdapat lebih dari satu route *resource*, dapat digabung menggunakan method `resources`.

```
Route::resources([
  'categories'=>CategoryController::class,
  'posts'=>PostController::class
]);
```

View Index

Buat file `admin/posts/index.blade.php` yang akan digunakan sebagai tampilan halaman index post.

```
@extends('layouts.app')
@section('content')
<div class="container">
  <div class="row justify-content-center">
    <div class="col-md-8">
      <div class="d-flex flex-row  mb-3">
        <a href="{{route('posts.create')}}" class="btn btn-primary">Tambah
Post</a>
      </div>
      <div class="card">
        <div class="card-header">Post</div>
        <div class="card-body">
          <table class="table table-bordered">
            <thead>
              <tr>
                <th>No</th>
                <th>Judul</th>
                <th>Kategori</th>
                <th>Isi</th>
                <th>Gambar</th>
                <th>Status</th>
                <th>User</th>
                <th>Aksi</th>
              </tr>
            </thead>
            <tbody>
            </tbody>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>
@endsection
```

Panggil view yang telah dibuat di method `index`

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{

  public function index()
  {
    return view('admin.posts.index');
  }
}
```

Silahkan buka di browser http://laravel-blog.test/admin/posts, maka akan muncul halaman index post.

Form Tambah Post

Buat file `admin/posts/create`, isi dengan kode berikut:

```
@extends('layouts.app')
@section('content')
<div class="container">
 <div class="row justify-content-center">
  <div class="col-md-8">

   <div class="card">
    <div class="card-header">Post Baru</div>

    <div class="card-body">
     <form method="POST" action="{{route('posts.store')}}"
enctype="multipart/form-data">
      @csrf
      <!-- Kode Lain -->
      <div class="mb-3">
       <label for="title" class="form-label">Judul</label>
       <input type="text" class="form-control @error('title') is-invalid
@enderror" name="title" id="title"
        value="{{old('title')}}">
       @error('title')
       <div class="invalid-feedback">
        {{$message}}
       </div>
       @enderror
      </div>
      <!-- Kode Lain -->

      <button type="submit" class="btn btn-primary">Submit</button>
     </form>
    </div>
   </div>
  </div>
 </div>
</div>
@endsection
```

> **Notice:** Full Source Code dapat diakses di sini

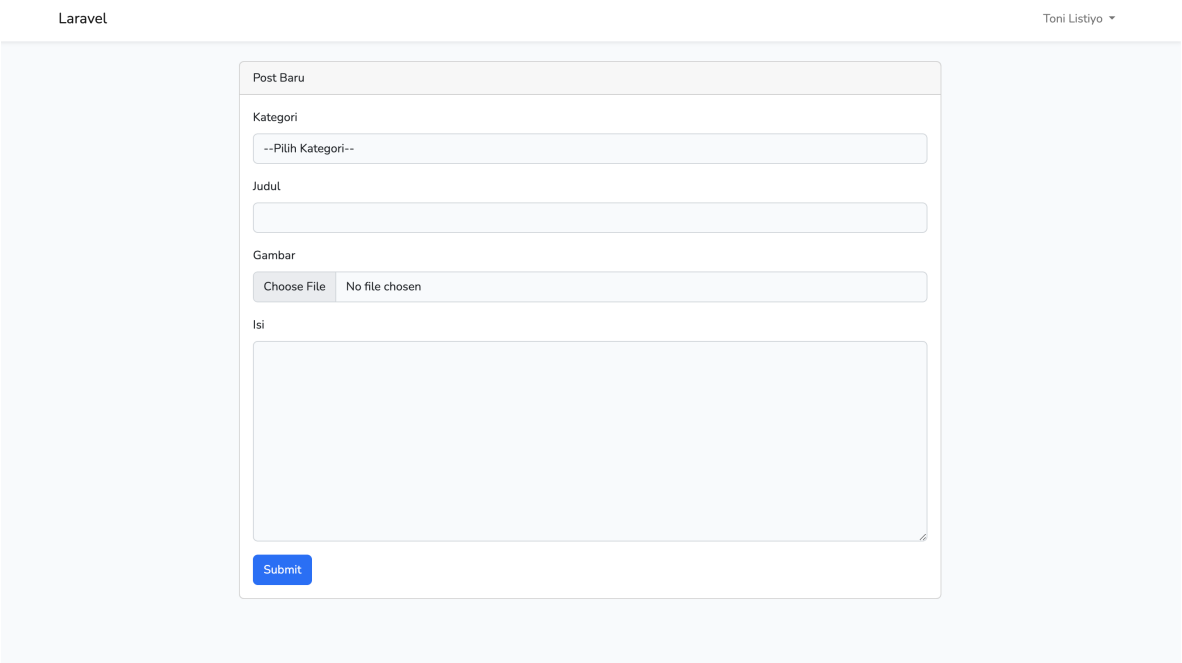
- `multipart/form-data` digunakan untuk mengirim data yang berisi file

- helper `old` digunakan untuk mengembalikan nilai yang sebelumnya diinput ketika terjadi kesalahan validasi pada form. Tanpa menambahkan helper ini, jika terjadi kesalahan input sebelumnya akan hilang.

Panggil view `create.blade.php` di method `create`

```
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use App\Models\Category;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
  public function create()
  {
    $categories = Category::all();
    return view('admin.posts.create',compact('categories'));
  }
}
```

Post membutuhkan kategori sehingga kita perlu mem-*passing* data kategori ke dalam view `create.blade.php` . Berikut adalah tampilan form tambah post.



Menyimpan Post ke Database

Proses untuk menyimpan post ke dalam database dilakukan di method `store`

```
<?php

namespace App\Http\Controllers\Admin;

use App\Models\Post;
use ReflectionClass;
use App\Models\Category;
use Illuminate\Support\Str;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
  public function store(Request $request)
  {
    $request->validate([
        'title' => 'required|unique:posts,title',
        'body'=>'required|min:50',
        'category_id'=>'required',
        'cover'=>'nullable|image'
    ]);
    Post::create([
      'title'=>$request->title,
      'body'=>$request->body,
      'slug'=>Str::slug($request->title),
      'category_id'=>$request->category_id,
      'status'=>StatusEnum::DRAFT,
      'user_id'=>auth()->id()
    ]);
    return redirect()->route('posts.index')->with('success', 'Post berhasil
disimpan');
  }

}
```

Jika kalian perhatikan, ada perbedaan dengan `CategoryController` saat mengambil nilai input dari form. Pada `CategoryController` kita menggunakan `$request->input()` , namun disini kita menggunakan `$request->{nama_field}` , ini dinamakan Dynamic Properties yang memungkinkan kita untuk mengakses properti objek dengan nama yang dinamis.

Untuk upload cover kita skip dulu, kita bahas di materi upload file.

Menampilkan Daftar Post

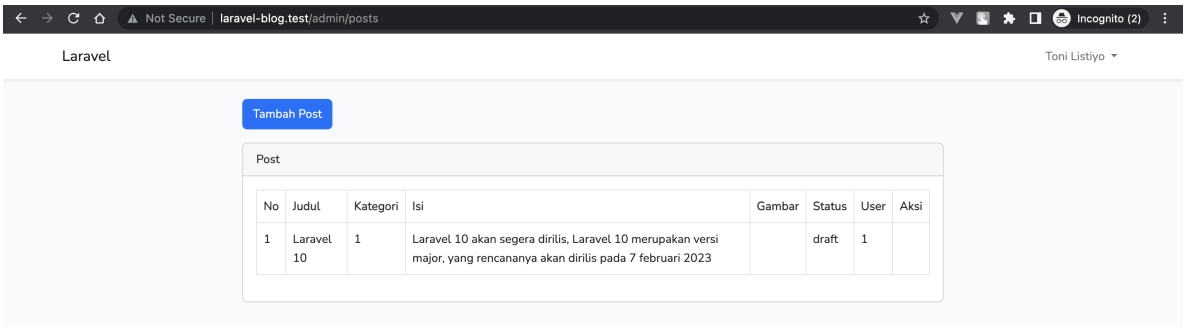
Setelah sukses manyimpan post ke database, kita perlu menampilkannya di halaman index post. Pada method `index`, ubah kodenya menjadi seperti berikut

```
<?php

namespace App\Http\Controllers\Admin;

// Kode Lain

class PostController extends Controller
{
  public function index()
  {
    $posts = Post::all();
    return view('admin.posts.index',compact('post'));
  }

}
```

Di view index tambahkan kode berikut di dalam tag `<tbody>`

```
@foreach ($posts as $post)
  <tr>
    <td>{{$loop->iteration}}</td>
    <td>{{$post->title}}</td>
    <td>{{$post->category_id}}</td>
    <td>{{$post->body}}</td>
    <td></td>
    <td>{{$post->status}}</td>
    <td>{{$post->user_id}}</td>
    <td>

    </td>
  </tr>
 @endforeach
```

Tampilan dari halaman index post akan seperti ini



Dapat kalian lihat pada kolom kategori dan user kita masih menampilkan id kategori dan id user, untuk menampilkan nama kategori dan nama user diperlukan relasi antara model *Post* dengan *Category* dan Post dengan *User* . Akan kita bahas di bab relasi.

Update Post

Tambahkan terlebih dahulu tombol aksi untuk menuju ke halaman edit post di tabel daftar post

```
<div class="d-flex flex-row">
  <a href="{{route('posts.edit',$post)}}" class="btn btn-warning">Edit</a>
</div>
```

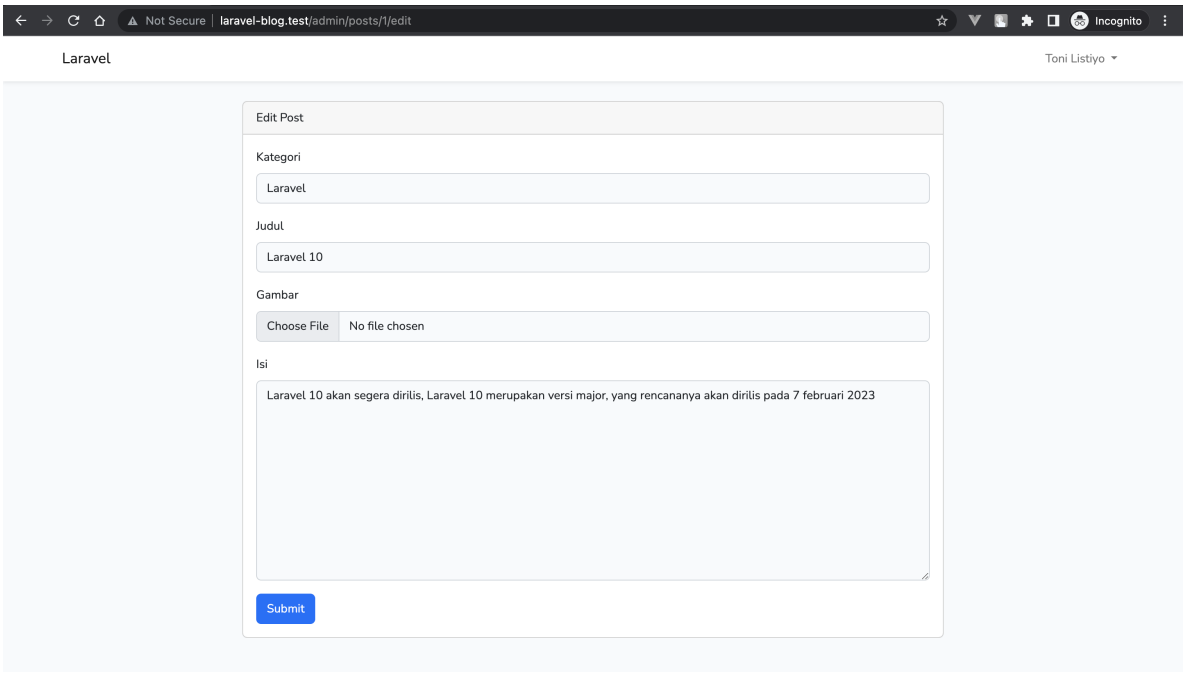
Kemudian buat file `admin/posts/edit.blade/php`

Source Code bisa dilihat di sini

Panggil view edit pada method `edit`

```
<?php

namespace App\Http\Controllers\Admin;

use App\Enums\StatusEnum;
use App\Models\Post;
use ReflectionClass;
use App\Models\Category;
use Illuminate\Support\Str;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
  public function edit(Post $post)
  {
   $categories = Category::all();
   return view('admin.posts.edit',compact('post','categories'));
  }
}
```

Ketika tombol edit diklik, maka akan muncul halaman edit post



Untuk menjalankan aksi update, ubah `method` update menjadi seperti berikut

```php
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PostController extends Controller
{
  // Kode lain
  public function update(Request $request, Post $post)
  {
    $request->validate([
      'title' => 'required|unique:posts,title,'.$post->id,
      'body' => 'required|min:50',
      'category_id' => 'required',
      'cover' => 'nullable|image'
    ]);
    $post->update([
      'title' => $request->title,
      'body' => $request->body,
      'slug' => Str::slug($request->title),
      'category_id' => $request->category_id,
    ]);
    return redirect()->route('posts.index')->with('success', 'Post berhasil diupdate');
  }

}
```

`$post->id` menunjukkan pada saat validasi, akan melewatkan ID dari postingan yang sedang diupdate sehingga validasi dapat mengabaikan rule unique untuk judul postingan yang sedang diupdate.

Tanpa parameter ini, validasi akan selalu gagal ketika kita mengupdate postingan karena judul postingan yang diupdate sudah ada di dalam database.

Hapus Post

Tambahkan tombol hapus di tabel daftar post

```
<div class="d-flex flex-row">
  <a href="{{route('posts.edit',$post)}}" class="btn btn-warning">Edit</a>
  <form action="{{ route('posts.destroy', $post) }}" method="POST">
    @csrf
    @method('DELETE')
    <button onclick="return confirm('Apakah anda yakin akan hapus?')"
class="btn btn-danger"
        type="submit">Hapus</button>
  </form>
</div>
```

Kemudian buat aksi untuk melakukan hapus di method `destroy`

```
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PostController extends Controller
{
  public function destroy(Post $post)
  {
    $post->delete();
    return redirect()->route('posts.index')->with('success', 'Post berhasil
dihapus');
  }
}
```

Membuat Navbar Link

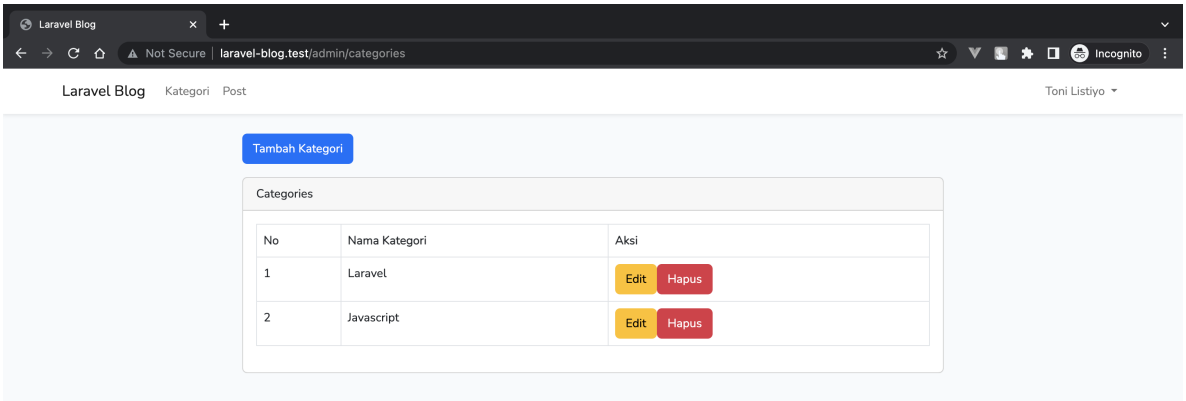
Sebelum mengakhiri fitur post, mari rapikan dulu web yang kita buat. Agar mudah mengkases menu kategori dan post, tambahkan link di navbar dengan menambahkannya di `resources/views/layouts/app.blade.php`

```
<!-- Left Side Of Navbar -->
<ul class="navbar-nav me-auto">
  <li class="nav-item">
      <a href="{{route('categories.index')}}" class="nav-link">Kategori</a>
  </li>
  <li class="nav-item">
      <a href="{{route('posts.index')}}" class="nav-link">Post</a>
  </li>
</ul>
```

Ubah juga nama aplikasi menjadi “Laravel Blog” di file `.env`

```
APP_NAME="Laravel Blog"
```

Sehingga tampilan halaman admin menjadi seperti ini :



One to Many Relationship

Hubungan antara model Category dengan Post adalah *one to many* artinya satu kategori dapat memiliki banyak post, namun sebuah post hanya dimiliki oleh satu kategori, begitupun dengan hubungan model User dengan Post , User dapat memiliki (membuat) banyak post, namun sebuah post hanya memiliki (boleh dibuat oleh) satu user.

Konfigurasi

Untuk membuat relasi *one to many* yaitu dengan menambahkan method `hasMany` di model *parent* dan menambahkan method `belongsTo` di model *child*.

User HasMany Post

Tambahkan method `hasMany()` di model `User` ini digunakan untuk menentukan bahwa model `User` memiliki banyak `Post`.

```php
<?php

namespace App\Models;

// Kode lain

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

Karena user memiliki banyak post, maka untuk penamaan method yang disarankan adalah bentuk plural (jamak) untuk post yaitu `posts`, nama lain bisa saja namun **tidak disarankan**.

> **Warning:** Perlu diingat dalam bahasa Inggris, ada beberapa benda jika diubah ke bentuk plural tidak hanya ditambahkan huruf `s` di akhir kata, contoh bentuk plural person bukan persons melainkan people, *mouse = mice, child = children* dll.

Untuk mengetahui bentuk plural suatu benda dapat menggunakan helper `Str::plural`. Untuk mencobanya bisa menggunakan tinker, jalankan `php artisan tinker` di terminal, ketik kode di bawah ini

```
> Str::plural('person')
= "people"
> Str::plural('child')
= "children"
```

Method `hasMany` memiliki tiga parameter yaitu :

1. `$related` : Nama child model yang akan dihubungkan dengan parent model. Parameter ini harus diisi dengan nama class child model yang akan dihubungkan. Dalam contoh ini adalah model `Post`.

2. `$foreignKey` : Nama kolom pada *child* tabel yang akan digunakan sebagai *foreign key* untuk menghubungkan *parent* model dan *child* model. Jika tidak diisi, maka Laravel akan menggunakan default *foreign key* yaitu nama *parent* model ditambah dengan `_id`. Dalam contoh ini, karena parameter ke dua tidak diisi maka secara *default* akan bernilai `user_id`.

3. `$localKey` : Nama kolom pada *parent* tabel yang digunakan sebagai kunci lokal / *primary key* untuk menghubungkan *parent* model dan *child* model. Jika tidak diisi, maka Laravel akan menggunakan default *primary key* yaitu `id`.

Parameter ke dua dan ke tiga perlu didefinisikan jika tidak mengikuti *convension* laravel, misalnya *foreign key* untuk user di tabel post adalah `id_user` maka parameter ke dua harus diisi dengan `id_user` .

Post belongTo User

Untuk konfigurasi di model `Post` yaitu dengan menambahkan method `belongsTo`, karena post hanya dimiliki oleh satu user maka untuk penamaan methodnya menggunakan bentuk tunggal dari parent model.

```php
<?php

namespace App\Models;

use App\Enums\StatusEnum;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Database\Eloquent\Factories\HasFactory;

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

Method `belongsTo` memiliki empat parameter yaitu :

1. `$related` : Parameter ini digunakan untuk menentukan nama class model terkait. Dalam contoh ini model yang berelasi adalah `User`.

2. `$foreignKey` (opsional): Parameter ini digunakan untuk menentukan nama kolom *foreign key* pada tabel model yang sedang didefinisikan. Secara default, Laravel akan menggunakan nama model terkait ditambah dengan `_id` sebagai *foreign key*. Dalam contoh ini maka nilainya adalah `user_id`

3. `$ownerKey` (opsional): Parameter ini digunakan untuk menentukan nama kolom *primary key* pada tabel model terkait. Secara default, Laravel akan menggunakan `id` sebagai kunci lokal.

4. `$relation` (opsional): Parameter ini digunakan untuk menentukan nama relasi yang akan digunakan. Jika tidak diisi, Laravel akan menggunakan nama class model terkait sebagai nama relasi. Dalam contoh ini maka nilainya adalah `user`.

Category HasMany Post

Penjelasan mengenai konfigurasi relasi *one to many* menggunakan Eloquent sudah dibahas di relasi Post dengan User. Jadi untuk relasi Category dengan Post langsung saja tambahkan method `hasMany` di model `Category` dan `belongsTo` di model `Post`

```php
<?php

namespace App\Models;

// Kode lain

class Category extends Model
{
  public function post(): HasMany
  {
    return $this->hasMany(Post::class);
  }
}
```

```php
<?php

namespace App\Models;

// Kode lain

class Post extends Model
{
  public function user(): BelongsTo
  {
    return $this->belongsTo(User::class);
  }

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

Menampilkan Relasi

Setelah method `hasMany` dibuat, kita dapat mengakses *collection* `posts` yang berelasi dengan `user` atau `category` menggunakan property `posts`.

> **Notice:** Ingat, Eloquent menyediakan *dynamic relationship property*, kita dapat mengakses method relationship seolah-olah mereka didefinisikan sebagai properti pada model

Sebagai contoh menampilkan `category` dan `post` yang berelasi:

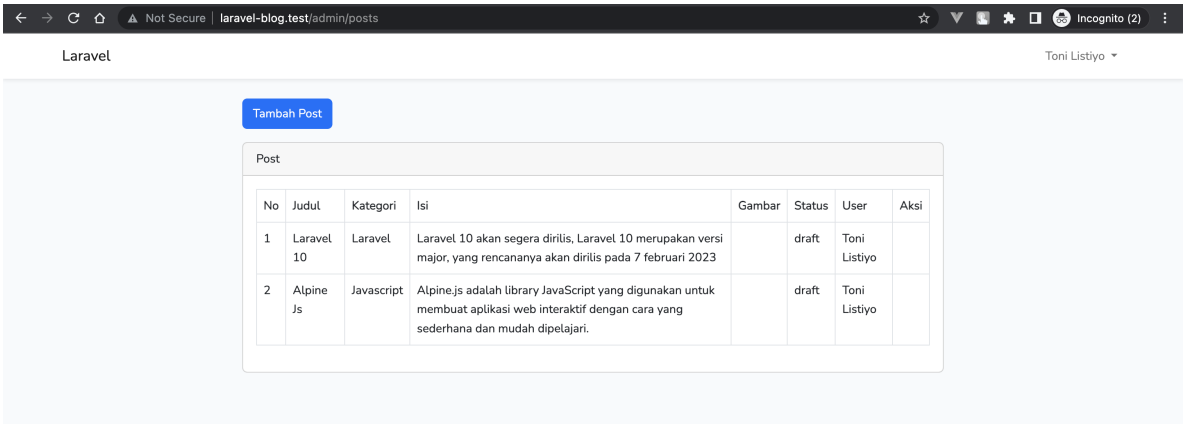
```
$category = Category::find(1);
$category->posts;
```

Sedangkan untuk menampilkan data `category` dan data `user` dari `posts` dapat memanggil property `category` (untuk mengakses data category) dan property `user` (mengakses properti `user`). Langsung saja kita implementasikan di studi kasus.

Ubah kode di `admin/posts/index.blade.php` menjadi seperti ini :

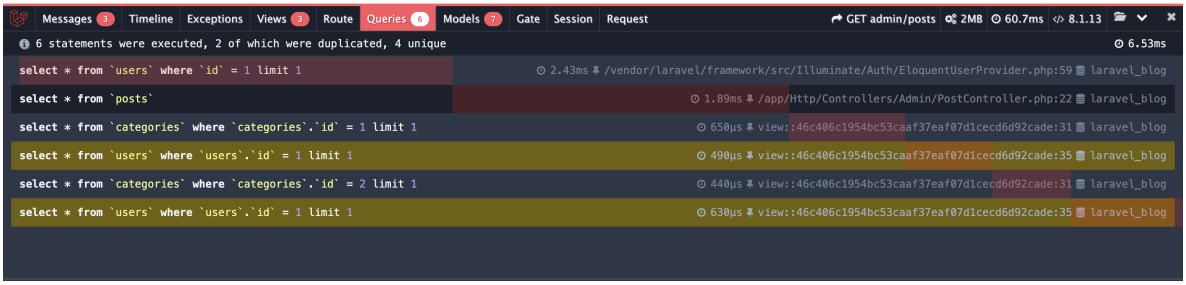
```
<!-- Kode lain -->
@foreach ($posts as $post)
 <tr>
  <td>{{$loop->iteration}}</td>
  <td>{{$post->title}}</td>

  <!-- post memanggil property category -->
  <td>{{$post->category->name}}</td>
  <td>{{$post->body}}</td>
  <td></td>
  <td>{{$post->status}}</td>

  <!-- post memanggil property user -->
  <td>{{$post->user->name}}</td>
  <td>

  </td>
 </tr>
@endforeach
<!-- kode lain -->
```

Sekarang nama kategori dan nama user sudah muncul di halaman index post.



N+1 Query Problem

Sekilas tidak ada yang salah dengan hasil diatas, nama kategori dan nama user sudah muncul namun jika kita cek query yang dieksekusi menggunakan Laravel Debugbar, terdapat warning 2 query duplikat , 4 unik.



Dua query yang duplikat itu disebut N+1 query problem, yaitu masalah yang terjadi ketika sebuah aplikasi web melakukan banyak permintaan ke database, terutama ketika melakukan relasi antar tabel, yang menyebabkan performa aplikasi menurun drastis.

Masalah ini sering terjadi pada aplikasi web yang menggunakan ORM (Object-Relational Mapping) seperti Laravel, karena ORM akan menghasilkan query secara otomatis untuk relasi antar tabel berdasarkan definisi model. Jika tidak diatasi, masalah ini dapat menyebabkan **performa aplikasi menurun dan waktu respon yang lambat serta konsumsi banyak memori**.

Contohnya, ketika sebuah aplikasi perlu menampilkan daftar post dan nama penulisnya, aplikasi tersebut mungkin akan mengambil data post dengan satu query (`SELECT * FROM posts`) dan kemudian mengambil data user dengan query terpisah untuk setiap post yang ada. Sehingga, jika ada 100 post, maka akan dilakukan 101 query (1 query untuk post, dan 100 query untuk user).

Dalam studi kasus ini, aplikasi menampilkan daftar post yang berjumlah 2 record (1 query telah dieksekusi), untuk setiap post, aplikasi juga menampilkan nama kategori post (2 query telah dieksekusi). Sampai disini aplikasi telah menjalankan 3 query, inilah kenapa dinamakan N+1. Aplikasi juga menampilkan penulis post untuk tiap post (2 query telah dieksekusi lagi), sehingga total menjalankan 5 query padahal seharusnya cukup mengkesekusi 3 query saja .

N+1 Query Detector

N+1 query detector adalah tool atau library yang digunakan untuk mendeteksi masalah N+1 query pada aplikasi web. Tool ini bekerja dengan cara menganalisis query yang dihasilkan oleh aplikasi dan mengidentifikasi query yang mungkin menyebabkan masalah N+1 query.

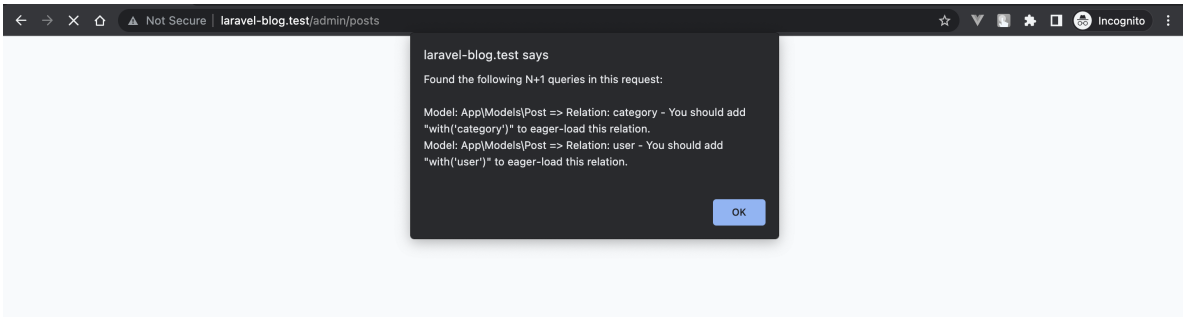
Laravel Query Detector

Salah satu librarynya adalah Laravel Query Detector, library ini akan menampilkan alert

apabila di aplikasi kita terdapat masalah N+1. Untuk instal dapat dilakukan melalui composer

```
composer require beyondcode/laravel-query-detector --dev
```

Setelah instalasi selesai refresh halaman index post, maka akan muncul alert ditemukan masalah N+1 beserta solusinya.



Disable Lazy Loading

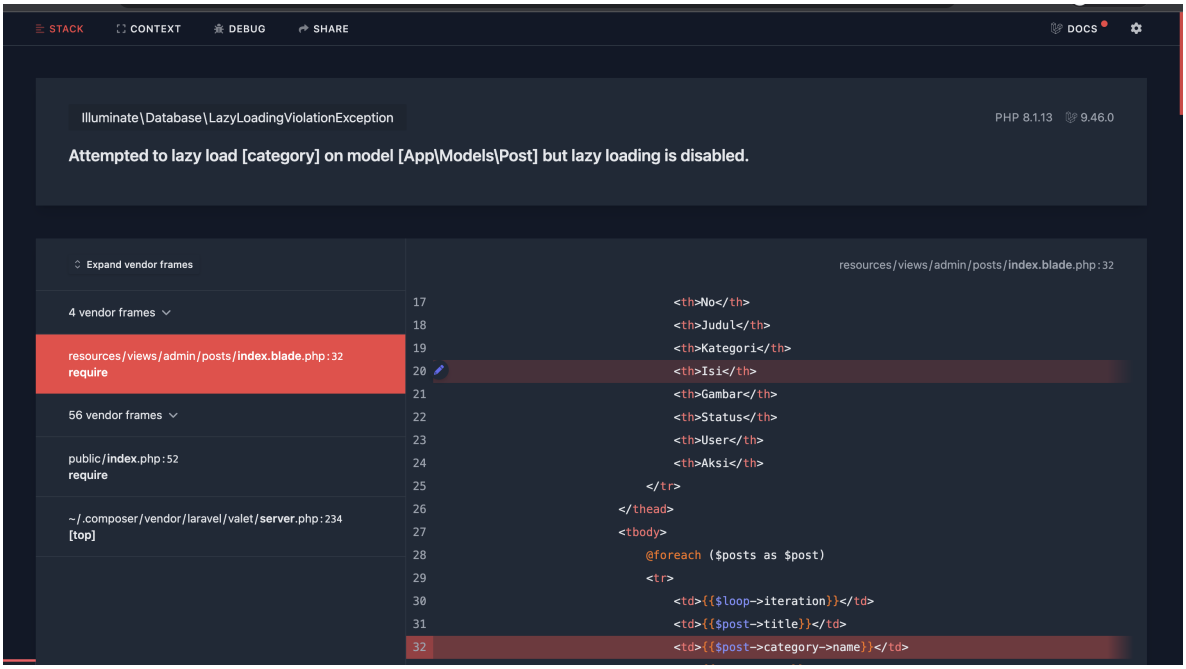
Cara kedua adalah dengan menggunakan method bawaan laravel yaitu `preventLazyLoading`. Cara ini lebih ketat dibanding cara pertama, karena akan menampilkan error jika terdapat masalah N+1.

Method ini akan membuat pengambilan data relasi antar tabel tidak lagi dilakukan secara otomatis, dan developer harus melakukan eager loading secara eksplisit menggunakan method `with` pada query builder. Hal ini dapat membantu mencegah terjadinya masalah N+1 query pada aplikasi web.

Method `Model::preventLazyLoading()` dapat dipanggil pada `AppServiceProvider` di method `boot`.

```php
<?php

namespace App\Providers;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
  public function boot()
  {
    Model::preventLazyLoading();
  }
}
```

Jika di aplikasi terdapat masalah N+1, aplikasi akan menampilkan error.



Solusi N+1

Solusi untuk masalah N+1 adalah dengan melakukan eager loading, yaitu mengambil data relasi antar tabel secara eksplisit pada saat query pertama dilakukan.

Di Laravel, terdapat beberapa cara untuk melakukan eager loading, yaitu:

1. Menggunakan method `with`

   Method ini digunakan untuk mengambil data relasi antar tabel secara eksplisit

pada saat query pertama dilakukan. Contoh:

```
$posts = Post::with(['user','category'])->get();
```

2. Menggunakan method `load`

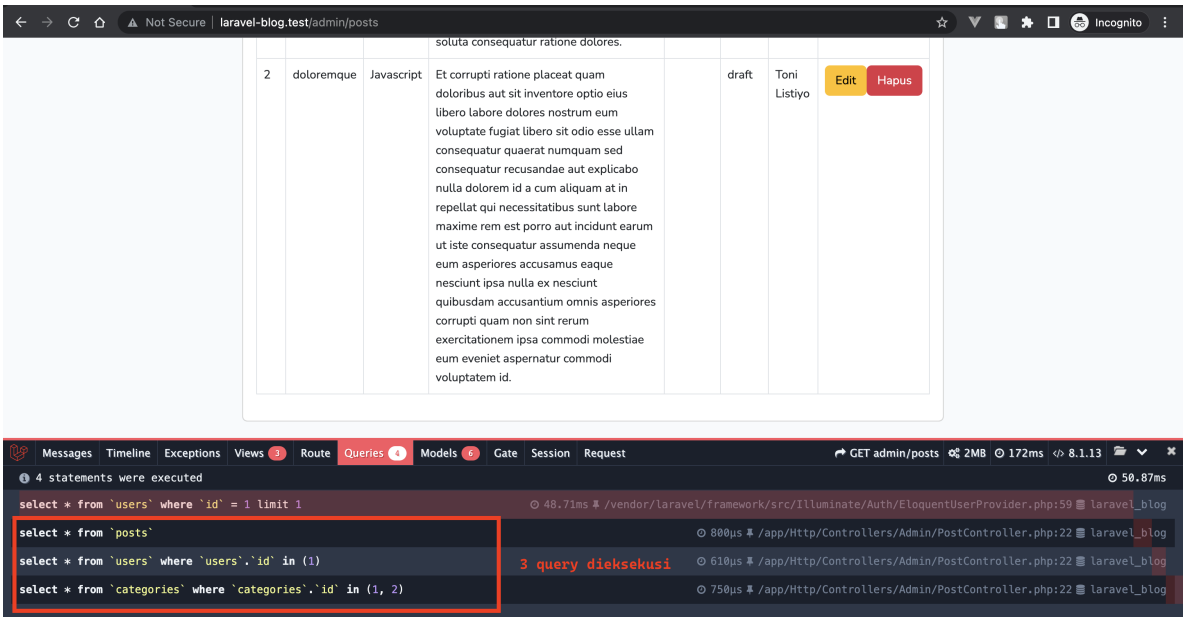
   Method ini digunakan untuk mengambil data relasi antar tabel setelah query pertama dilakukan. Contoh

```
$posts = Post::all();
$post->load(['user','category'])
```

Implementasi untuk studi kasus aplikasi yang kita buat seperti ini:

```
<?php

namespace App\Http\Controllers\Admin;

// Kode Lain

class PostController extends Controller
{
  public function index()
  {
    $posts = Post::with(['user','category'])->get();
    return view('admin.posts.index',compact('posts'));
  }

}

// Kode lain
```

Hasilnya sekarang query yang dieksekusi hanya 3 saja. Berapapun jumlah post, query yang akan dieksekusi tetap 3.


Not Secure | laravel-blog.test/admin/posts
Incognito
soluta consequatur ratione dolores.
2
doloremque
Javascript
Et corrupti ratione placeat quam doloribus aut sit inventore optio eius libero labore dolores nostrum eum voluptate fugiat libero sit odio esse ullam consequatur quaerat numquam sed consequatur recusandae aut explicabo nulla dolorem id a cum aliquam at in repellat qui necessitatibus sunt labore maxime rem est porro aut incidunt earum ut iste consequatur assumenda neque eum asperiores accusamus eaque nesciunt ipsa nulla ex nesciunt quibusdam accusantium omnis asperiores corrupti quam non sint rerum exercitationem ipsa commodi molestiae eum eveniet aspernatur commodi voluptatem id.
draft
Toni Listiyo
Edit
Hapus
Messages
Timeline
Exceptions
Views 3
Route
Queries 4
Models 6
Gate
Session
Request
GET admin/posts
2MB
172ms
8.1.13
4 statements were executed
50.87ms
select * from `users` where `id` = 1 limit 1
48.71ms /vendor/laravel/framework/src/Illuminate/Auth/EloquentUserProvider.php:59 laravel_blog
select * from `posts`
800µs /app/Http/Controllers/Admin/PostController.php:22 laravel_blog
select * from `users` where `users`.`id` in (1)
610µs /app/Http/Controllers/Admin/PostController.php:22 laravel_blog
select * from `categories` where `categories`.`id` in (1, 2)
750µs /app/Http/Controllers/Admin/PostController.php:22 laravel_blog
3 query dieksekusi

Upload Gambar

Untuk mengelola file dan direktori di dalam aplikasi, Laravel menyediakan library Flysistem. Flysystem dapat digunakan untuk menyimpan file di berbagai sistem penyimpanan, termasuk sistem file lokal, Amazon S3, Google Cloud Storage, Dropbox, dan banyak lagi. Dengan menggunakan Flysystem dalam Laravel, Kita dapat menyimpan dan mengelola file dalam aplikasi dengan mudah.

Konfigurasi Filesystem

Biasanya file yang diupload oleh user sebaiknya disimpan di folder `storage` dan dipindahkan ke folder `public` untuk di akses di frontend. Namun bisa juga di simpan di folder `public` jika tidak ada masalah dengan keamanan.

Untuk studi kasus ini, kita akan membuat *custom disk* dengan nama **cover** sebagai lokasi menyimpan gambar cover. Untuk membuat *custom disk* dapat dilakukan di file `config/filesystems.php` berikut kodenya :

```
// kode lain
return [

  'disks' => [
    'covers' => [
      'driver' => 'local',
      'root' => storage_path('app/covers'),
      'url' => env('APP_URL') . '/covers',
      'visibility' => 'public',
      'throw' => false,
    ],
  ],
];

// kode lain
```

Penjelasan konfigurasi di atas adalah sebagai berikut:

- **'driver'** => `'local'` : Menunjukkan bahwa driver yang digunakan adalah local, yaitu sistem penyimpanan lokal di server.
- **'root'** => `storage_path('app/covers')` : Menunjukkan bahwa file-file yang diunggah ke disk ini akan disimpan di folder `storage/app/covers` pada server. Oleh karena itu kita perlu membuat juga folder covers.
- **'url'** => `env('APP_URL') . '/covers'` : Menunjukkan URL publik yang dapat digunakan untuk mengakses file-file di disk ini. Dalam hal ini, URL yang digunakan adalah APP_URL/covers.
- **'visibility'** => 'public' : Menunjukkan bahwa file-file yang disimpan di disk ini dapat diakses oleh publik.
- **'throw'** => 'false' : Menunjukkan bahwa ketika terjadi error dalam proses upload file, error tersebut tidak akan dilempar ke pengguna, melainkan akan diabaikan.

Dengan konfigurasi tersebut, disk "covers" sudah siap digunakan untuk mengunggah dan mengelola file gambar yang akan digunakan sebagai cover post di aplikasi yang kita buat.

Upload Cover Post

Tambahkan kode untuk upload gambar di `PostController::store` , sehingga kodenya akan seperti ini:

```
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PostController extends Controller
{

  // Kode lain
  public function store(Request $request)
  {

    $request->validate([
      'title' => 'required|unique:posts,title',
      'body'=>'required|min:50',
      'category_id'=>'required',
      'cover'=>'nullable|image'
    ]);
    if ($request->hasFile('cover')) {
      $cover = $request->file('cover')->store('/', 'covers');
    }

    Post::create([
      'title'=>$request->title,
      'body'=>$request->body,
      'slug'=>Str::slug($request->title),
      'category_id'=>$request->category_id,
      'status'=>StatusEnum::DRAFT,
      'cover'=>$cover??null,
      'user_id'=>auth()->id()
    ]);
    return redirect()->route('posts.index')->with('success', 'Post berhasil
disimpan');
  }
}
```

Penjelasan kode tersebut sebagai berikut:

`$request->file('cover')` : Mencari file dengan nama "cover" yang dikirim melalui request HTTP (melalui form). File ini harus memiliki tipe MIME yang diizinkan untuk diunggah di aplikasi.

`->store('/', 'covers')` : Memanggil metode `store()` untuk menyimpan file ke dalam disk **"covers"**. Tanda '/' di sini menunjukkan bahwa file akan disimpan di root directory (folder utama) di dalam disk **"covers"**.

`$request->file('cover')->store('/', 'covers')` : Akan menghasilkan nama file dalam bentuk random hash, menurut dokumentasi laravel, penamaan dengan cara seperti ini lebih aman dibanding dengan menggunakan nama file yang diupload.

Menampilkan Gambar

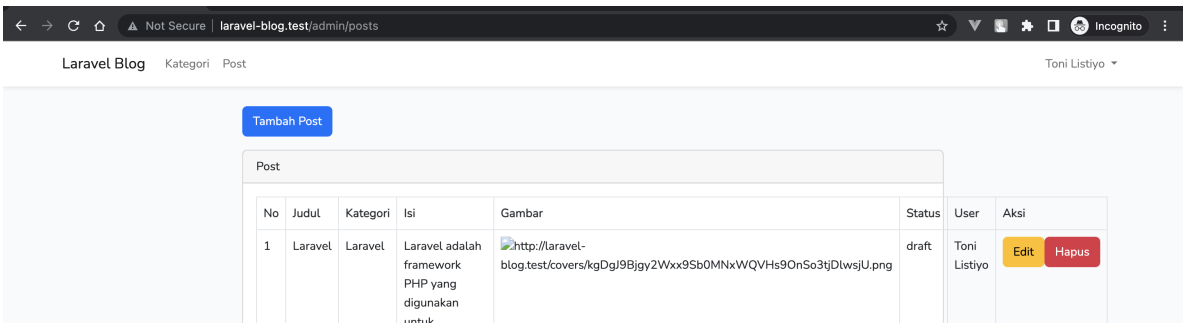
Untuk menampilkan gambar yang sudah diupload, buat method baru di Post model dengan nama `coverUrl`.

```
<?php

namespace App\Models;

// Kode Lain
use Illuminate\Support\Facades\Storage;

class Post extends Model
{
  // Kode Lain
  public function coverUrl()
  {
    return Storage::disk('covers')->url($this->cover);
  }
}
```

Seperti yang sudah dijelaskan di konfigurasi filesystem, method `url` merupakan URL publik yang dapat digunakan untuk mengakses file-file di disk **"covers"**.

Untuk menampilkan gambar cover, method `coverUrl()` ini bisa dipanggil pada view `admin/posts/index.blade.php` untuk mengambil URL gambar cover dari suatu post. Berikut ini cara menampilkan di tabel daftar post.

```
<img class="img-fluid" src="{{$post->coverUrl()}}" alt="{{$post->coverUrl()}}"
srcset="{{$post->coverUrl()}}">
```



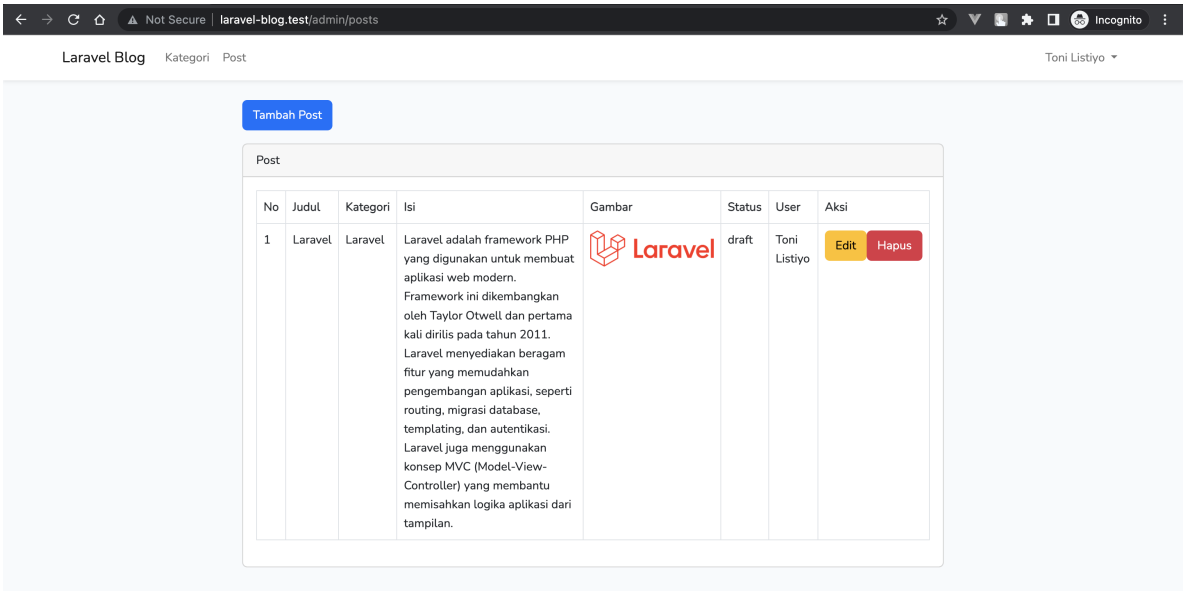
Ups. . gambar tidak muncul, hal ini disebabkan karena gambar berada di folder `storage` . File - file yang disimpan di folder ini tidak diakses secara langsung dari luar aplikasi. Agar dapat diakses publik kita perlu membuat link ke folder `public` .

Tambahkan kode berikut di file `config/filesystem.php`

```
<?php

return [

  'links' => [
      public_path('covers') => storage_path('app/covers'),
  ],

];
```

Kode tersebut berfungsi untuk menghubungkan folder `'covers'` yang berada di folder `'public'` dengan folder `'covers'` yang berada di folder `'storage'`. Hal ini memungkinkan kita untuk mengunggah file gambar pada folder `'public/covers'` dan menyimpannya pada folder `'storage/app/covers'`.

Jalankan `php artisan storage:link` , refresh halaman post index dan sekarang gambar cover sudah muncul.

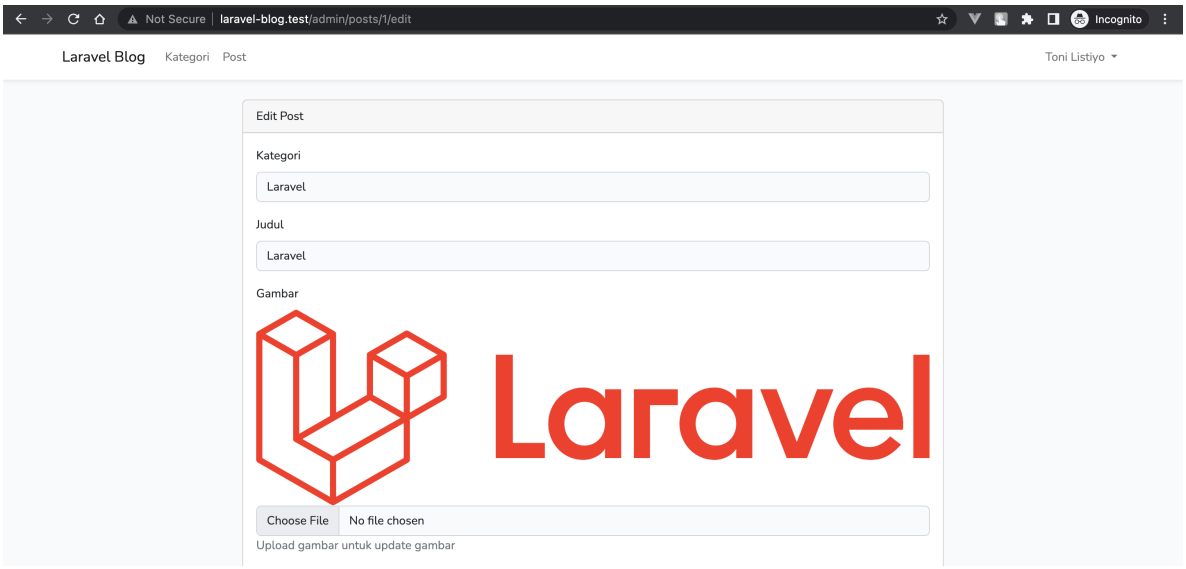


Update Gambar

Jika kita ingin mengupdate post beserta gambar covernya, maka kita perlu menghapus cover sebelumnya terlebih dahulu agar tidak memenuhi storage. Pertama-tama, ubah view edit dengan menambahkan field upload.

```
<div class="mb-3">
  <label for="cover" class="form-label">Gambar</label>
    <img src="{{$post->coverUrl()}}" class="img-fluid" alt="" srcset="">
  <input type="file" name="cover" class="form-control @error('cover') is-
invalid @enderror">
    <span class="text-muted">Upload gambar untuk update gambar</span>
   @error('cover')
   <div class="invalid-feedback">
    {{$message}}
   </div>
   @enderror

</div>
```

Pada saat edit, gambar cover akan muncul



Di method `update` tambahkan kode untuk upload gambar baru dan hapus gambar sebelumnya, serta tambahkan nama file yang akan disimpan ke database.

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Enums\StatusEnum;
use App\Models\Post;
use ReflectionClass;
use App\Models\Category;
use Illuminate\Support\Str;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Storage;

class PostController extends Controller
{
  public function update(Request $request, Post $post)
  {
    $request->validate([
      'title' => 'required|unique:posts,title,'.$post->id,
      'body' => 'required|min:50',
      'category_id' => 'required',
      'cover' => 'nullable|image'
    ]);
    if ($request->hasFile('cover')) {
      $cover = $request->file('cover')->store('/', 'covers');
      //hapus gambar sebelumnya
      Storage::disk('covers')->delete($post->cover);
    }
    $post->update([
      'title' => $request->title,
      'body' => $request->body,
      'slug' => Str::slug($request->title),
      'cover'=>$cover??$post->cover,
      'category_id' => $request->category_id,
    ]);
    return redirect()->route('posts.index')->with('success', 'Post berhasil
diupdate');
  }

}
```

Hapus Gambar

Ketika sebuah post dihapus, gambar cover seharusnya juga akan ikut terhapus. Ubah kode pada method `destroy` menjadi seperti berikut:

```
<?php

namespace App\Http\Controllers\Admin;

// Kode Lain

class PostController extends Controller
{
  // Kode Lain
  public function destroy(Post $post)
  {
    if($post->delete()){
      Storage::disk('covers')->delete($post->cover);
      return redirect()->route('posts.index')->with('success', 'Post berhasil
dihapus');
    }

  }
}
```

Pagination

Pagination di Laravel adalah fitur untuk membagi hasil query menjadi beberapa halaman sehingga memudahkan pengguna untuk menavigasi data dalam jumlah yang terukur. Fitur ini sangat penting untuk menghindari waktu load yang lama dan membuat aplikasi lebih responsif. Laravel menyediakan class Pagination untuk memudahkan pengguna membuat halaman paginasi pada aplikasi web.

Membuat Factory

Untuk implementasi pagination tentu kita perlu insert setidaknya 11 data, agar tidak melakukannya satu per satu, kita perlu memanfaatkan fitur factory di laravel.

Factory digunakan untuk memudahkan proses pengisian data awal atau seeding pada database. Factory berfungsi untuk membuat data palsu atau dummy data dengan struktur yang telah ditentukan sebelumnya.

Buat factory untuk `Category` model

```
php artisan make:factory CategoryFactory -m=Category
```

Pada file `CategoryFactory` tambahkan array untuk mengisi kategori ke database

```
<?php

namespace Database\Factories;

use Illuminate\Support\Str;
use Illuminate\Database\Eloquent\Factories\Factory;
class CategoryFactory extends Factory
{

  public function definition()
  {
    $name = fake()->word();
    return [
      'name'=>$name,
      'slug'=>Str::slug($name)
    ];
  }
}
```

Dengan menggunakan tinker, jalankan perintah untuk membuat kategori sebanyak 15 data

```
Category::factory(15)->create();
```

Pada file `CategoryController` gunakan function `paginate` untuk mendapatkan data kategori

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Models\Category;
use Illuminate\Support\Str;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class CategoryController extends Controller
{
  public function index()
  {
    $categories = Category::paginate(10);
    return view('admin.categories.index',compact('categories'));
  }

}
```

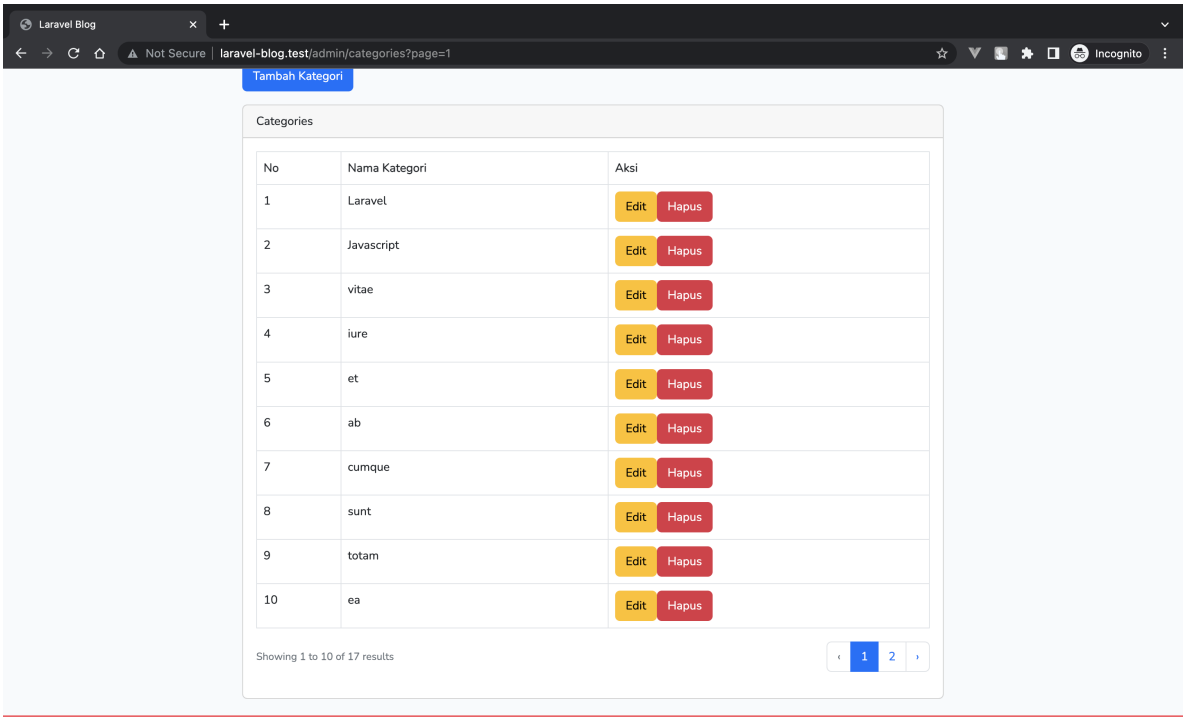
Secara default, style pagination menggunakan tailwind , agar support dengan Bootstrap 5 panggil method `useBootstrapFive` di method boot di `App\Providers\AppServiceProvider`

```php
<?php

namespace App\Providers;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Pagination\Paginator;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{

  public function boot()
  {
    Paginator::useBootstrapFive();
  }
}
```

Di view `admin/categories/index` , panggil `$categories->links()` untuk menampilkan pagination, dan no urut ubah menjadi `$categories->firstItem()+$loop->index` .

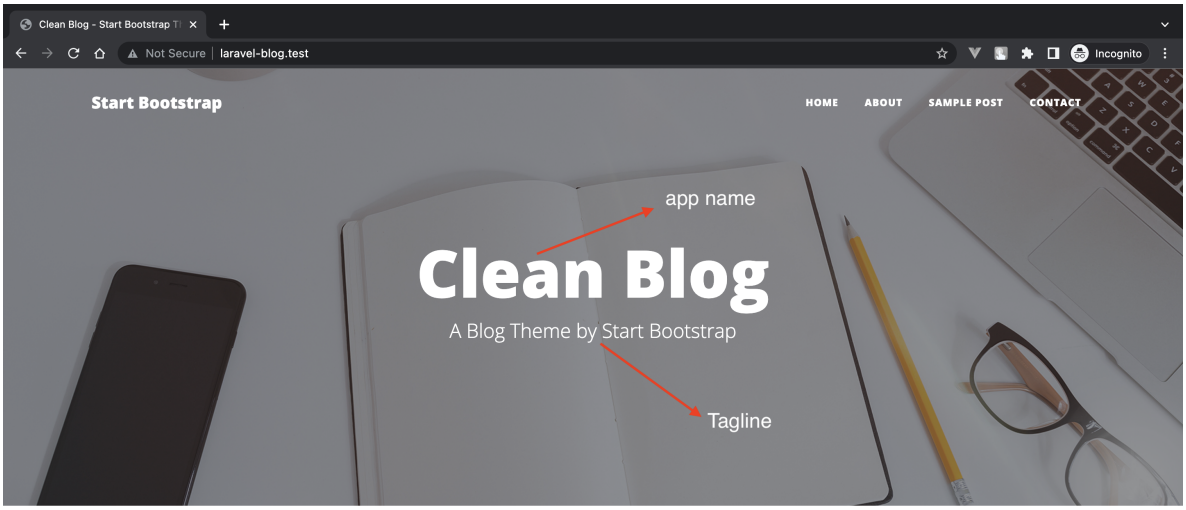
```
<!-- Kode lain -->
  <tbody>
    @foreach ($categories as $category)
      <tr>
        <td>{{$categories->firstItem()+$loop->index}}</td>
      </tr>
    @endforeach
  </tbody>
</table>
{{$categories->links()}}
```

Berikut adalah tampilan pagination di halaman index kategori



Fitur Setting

Fitur setting digunakan untuk mengatur nama aplikasi, tagline, dan banner. Pengguna hanya dapat menambahkan sekali saja kemudian mengupdatenya. Isi dari setting akan ditampilkan di halaman blog.



Man must explore, and this is exploration at its greatest

Problems look mighty small from 150 miles up

Posted by Start Bootstrap on September 24, 2022

Model dan Migration

Seperti biasa buat terlebih dahulu model dan migration.

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

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
  public function up()
  {
    Schema::create('settings', function (Blueprint $table) {
      $table->id();
      $table->string('app_name',20);
      $table->string('tagline',50);
      $table->string('banner')->nullable();
      $table->timestamps();
    });
  }

};
```

Route

Halaman Setting hanya akan memiliki satu halaman saja yaitu berupa form untuk membuat dan memperbarui setting. Method yang akan digunakan dalam `SettingController` hanya `index` dan `store`. Route hanya akan mengaktifkan kedua method ini dengan menambahkan metode `only` . SettingController akan kita buat setelah ini.

```
// Kode lain
Route::group(['prefix'=>'admin','middleware'=>'auth'],function(){
  Route::resources([
    'categories'=>CategoryController::class,
    'posts'=>PostController::class,
  ]);
Route::resource('setting',SettingController::class)->only(['index','store']);
});
```

View

Buat file `admin/setting.blade.php` sebagai tampilan untuk menambah atau memperbarui setting

```
@extends('layouts.app')
@section('content')
<div class="container">
 <div class="row justify-content-center">
  <div class="col-md-8">
   <div class="card">
    <div class="card-header">Setting</div>
    <div class="card-body">
     <form method="POST" action="{{route('setting.store')}}"
enctype="multipart/form-data">
      @csrf
      <input type="hidden" name="id" value="{{$setting?->id}}">
      <div class="mb-3">
       <label for="app_name" class="form-label">Nama Aplikasi</label>
       <input type="text" class="form-control @error('app_name') is-invalid
@enderror" name="app_name"
        id="app_name" value="{{$setting?->app_name}}">
       @error('app_name')
       <div class="invalid-feedback">
        {{$message}}
       </div>
       @enderror
      </div>

    // Kode Lain
     </form>
    </div>
   </div>
  </div>
 </div>
</div>
@endsection
```

Full Source Code dapat dilihat di sini

`?->` adalah *Nullsafe Operator* merupakan operator baru di PHP 8 yang memungkinkan melakukan operasi terhadap objek tanpa memeriksa apakah objek itu null terlebih dahulu.

Controller

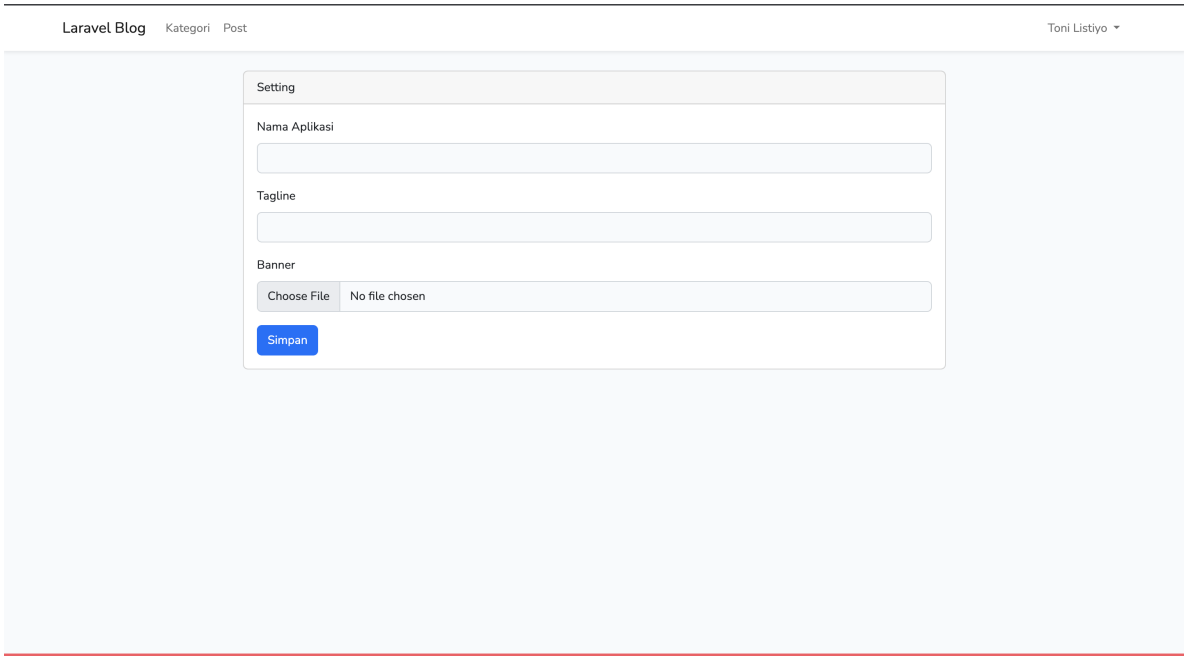
Buat controller untuk menangani tambah atau update setting.

```
php artisan make:controller Admin/SettingController -r
```

Pada method index panggil view `admin/setting.blade.php` , kirim juga data setting yang diambil dari database.

```
<?php

namespace App\Http\Controllers\Admin;

use App\Models\Setting;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class SettingController extends Controller
{
  public function index()
  {
    $setting = Setting::first();
    return view('admin.setting',compact('setting'));
  }
}
```

Berikut adalah tampilan form setting


Laravel Blog
Kategori
Post
Toni Listiyo
Setting
Nama Aplikasi
Tagline
Banner
Choose File
No file chosen
Simpan

Konfigurasi Upload Banner

Membuat konfigurasi upload banner sama dengan konfigurasi upload covers, kita buat custom disk dengan nama banner. Kemudian buat links untuk menghubungkan storage dengan public

```
<?php

return [
  'disks' => [
    //  Kode lain
    'banner' => [
        'driver' => 'local',
        'root' => storage_path('app/banner'),
        'url' => env('APP_URL') . '/banner',
        'visibility' => 'public',
        'throw' => false,
    ],

    // Kode lain

  ],

  'links' => [
    public_path('covers') => storage_path('app/covers'),
    public_path('banner') => storage_path('app/banner'),

  ],

];
```

> **Warning:** Jangan lupa untuk membuat folder banner didalam folder storage, dan jalankan `php artisan storage:link`

Create and Update Setting

Method store pada `SettingController` digunakan untuk menyimpan atau memperbarui data pada tabel settings. Jika data belum ada di tabel, method ini akan menyimpan data baru, dan jika sudah ada, method ini akan melakukan update pada data tersebut.

```
<?php

namespace App\Http\Controllers\Admin;

use App\Models\Setting;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use Exception;
use Illuminate\Support\Facades\Storage;

class SettingController extends Controller
{
  public function store(Request $request)
  {

    $request->validate([
      'app_name'=>'required|max:20',
      'tagline'=>'required|max:50',
      'banner'=>'nullable|image'
    ]);

    if ($request->hasFile('banner')) {
      Storage::deleteDirectory('banner');//menghapus semua file di direktori
banner
      $banner = $request->file('banner')->store('/', 'banner');
    }

    $setting = Setting::findOr($request->id,function()use($request,$banner){
      Setting::create([
        'app_name' => $request->app_name,
        'tagline' => $request->tagline,
        'banner' => $banner
      ]);
    });

    $setting->update([
      'app_name' => $request->app_name,
      'tagline' => $request->tagline,
      'banner' => $banner ?: $setting->banner
```

```
        ]);
        return redirect()->route('setting.index')->with('success', 'Setting
    berhasil diupdate');
      }

    }
```

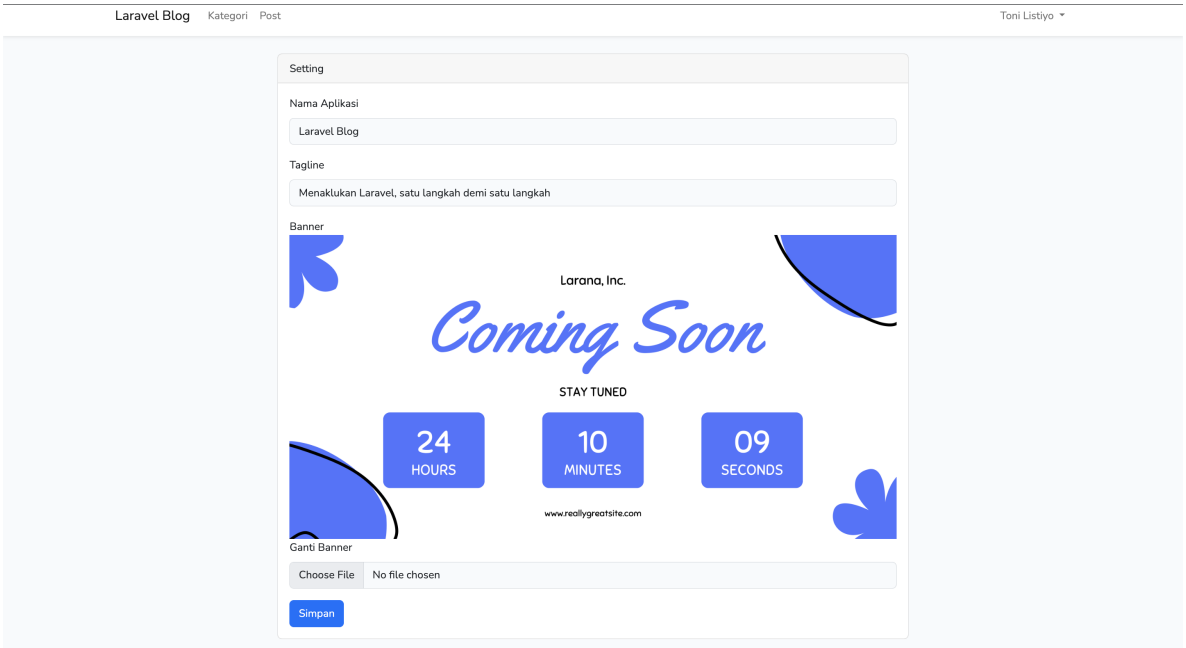
Method `findOr` akan mengembalikan instance model jika ditemukan, jika tidak maka akan mengeksekusi kode yang berada didalam *closure*.

Setting Model

Tambahkan property `guarded` di `Setting` model agar dapat melakukan mass assignment, tambahkan juga method untuk menampilkan gambar banner.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Storage;

class Setting extends Model
{
  use HasFactory;
  protected $guarded = [];

  public function bannerUrl(){
    return Storage::disk('banner')->url($this->banner);
  }
}
```

Silakan coba menambahkan data baru, dan berikut adalah hasil yang diharapkan jika proses berhasil dilakukan.



> **Warning:** Jangan lupa untuk menambahkan navbar link untuk mengakses halaman setting.

Homepage

Data yang sudah kita kelola melalui bagian admin selanjutnya akan ditampilkan di halaman depan blog, sehingga membuat tampilan blog menjadi lebih dinamis. Kita mulai dengan menampilkan setting aplikasi. Pada route `web.php` hapus kode ini

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

Kemudian ubah path `HomeController` menjadi `/`

```
Route::get('/', [App\Http\Controllers\HomeController::class,
'index'])->name('home');
```

Ubah kode `HomeController` menjadi seperti ini

```
<?php

namespace App\Http\Controllers;

use App\Models\Setting;
use Illuminate\Http\Request;

class HomeController extends Controller
{
  public function index()
  {
    $setting = Setting::first();
    return view('welcome',compact('setting'));
  }
}
```

Kode diatas kita mengambil data setting untuk ditampilkan ke view `welcome.blade.php`.

Pada file `welcome.blade.php` kirim variable `$setting` ke `frontend` layout melalui property `:setting`

```
<x-frontend :setting="$setting">
    <!-- Kode lain -->
</x-frontend>
```

Prefix `:` digunakan untuk mengirim variable ke *component* melalui attribut

Title Bar

Untuk mengubah title bar update di file `front/head.blade.php`

```
<head>
   <!-- Kode lain -->
    <title>{{$setting->app_name}} - {{$setting->tagline}}</title>
   <!-- Kode lain -->
</head>
```

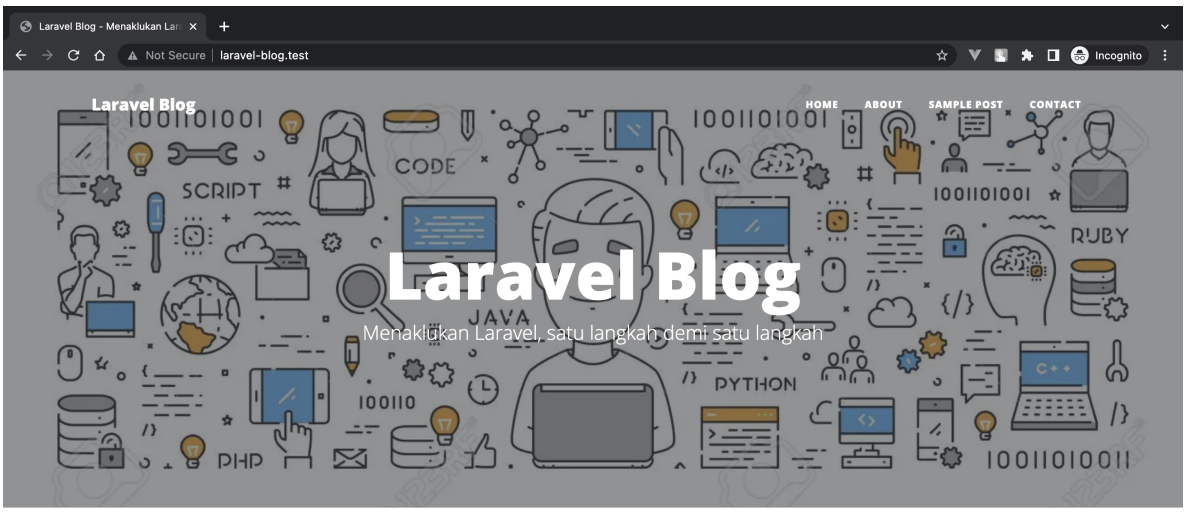
Variabel `$setting` merupakan property yang dikirim melalui `<x-frontend` component.

Banner

Untuk mengubah banner, title blog dan tagline update di file `front/header.blade.php`

```
<header class="masthead" style="background-image:
url({{$setting->bannerUrl()}})">
  <div class="container position-relative px-4 px-lg-5">
    <div class="row gx-4 gx-lg-5 justify-content-center">
      <div class="col-md-10 col-lg-8 col-xl-7">
        <div class="site-heading">
          <h1>{{$setting->app_name}}</h1>
          <span class="subheading">{{$setting->tagline}}</span>
        </div>
      </div>
    </div>
  </div>
</header>
```

Hasilnya halaman homepage akan seperti ini



Navbar

Navbar adalah sebuah elemen user interface (UI) yang bertujuan untuk memberikan navigasi dan akses cepat bagi pengguna untuk menavigasi situs web. Navbar biasanya berada di bagian atas halaman web dan merupakan bagian dari header website. Navbar menyediakan link utama ke halaman seperti halaman home, tentang kami, dan kontak. Dalam studi kasus ini, akan digunakan dua navbar yaitu navbar home dan navbar kategori. Navbar kategori akan berupa dropdown menu yang memudahkan pengguna untuk memilih kategori post yang ingin dilihat.

Ubah file `front/nav.blade.php` , di sini kita menampilkan kategori yang diambil dari database

```
<!-- resources/views/front/nav.blade.php -->
<!-- Kode Lain -->
<ul class="dropdown-menu border border-0">
  @foreach ($categories as $category)
    <li>
      <a class="dropdown-item" href="#">
        {{$category->name}}
      </a>
    </li>
  @endforeach
</ul>
<!-- Kode Lain -->
```

Full Source Code dapat dilihat di sini

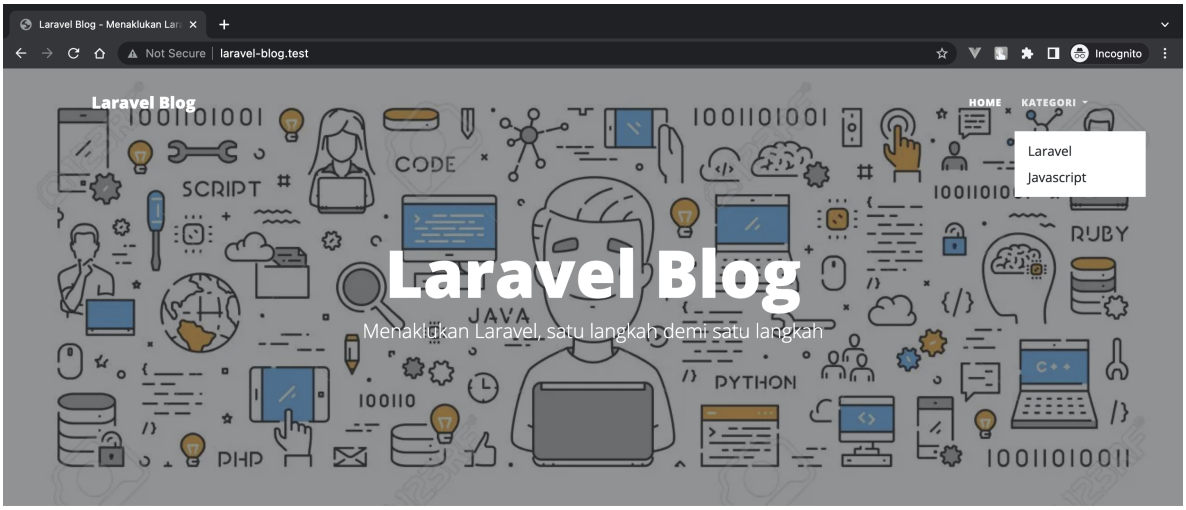
Pada `HomeController` , kirim data kategori ke view `welcome.blade.php`

```php
<?php

namespace App\Http\Controllers;

use App\Models\Category;
use App\Models\Setting;
use Illuminate\Http\Request;

class HomeController extends Controller
{
  public function index()
  {
    $setting = Setting::first();
    $categories = Category::select(['id','name','slug'])->get();
    return view('welcome',compact('setting','categories'));
  }
}
```

Di view `welcome.blade.php` kirim data kategori ke component `frontend`

```php
<x-frontend :setting="$setting" :categories="$categories">
  <!-- Kode lain -->
</x-frontend>
```

Hasilnya tampilan navbar akan seperti ini



Fix Error Setelah Login

Ketika login, akan muncul halaman error 404 not found, hal ini dikarenakan kita telah memodifikasi path untuk route home yang semula untuk dashboard admin menjadi halaman homepage. Untuk memperbaikinya buat controller untuk dashboard admin.

```
php artisan make:controller Admin/DashboardController -i
```

Buat route yang mengarah ke `Admin/DashboardController`

```
<?php

// Kode lain

use App\Http\Controllers\Admin\DashboardController;

// Kode lain

Route::group(['prefix'=>'admin','middleware'=>'auth'],function(){
  // Kode lain
  Route::get('dashboard',DashboardController::class)->name('dashboard');
});
```

Panggil view `home.blade.php` di `DashboardController`

```
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class DashboardController extends Controller
{
  public function __invoke(Request $request)
  {
    return view('home');
  }
}
```

Agar setelah login mengarah ke `DashboardController` , ubah nilai konstanta `HOME` di file

`app/Providers/RouteServiceProvider.php` menjadi route `admin/dashboard`

```
namespace App\Providers;

// Kode lain

class RouteServiceProvider extends ServiceProvider
{
  public const HOME = 'admin/dashboard';
  // Kode lain
}
```

Dengan begitu ketika login halaman akan mengarah ke path `admin/dashboard`

Halaman Post

Setelah kita memiliki data posting, kita perlu untuk menampilkannya di halaman blog agar dapat dibaca oleh pengunjung.

Post di Homepage

Untuk menampilkan post di homepage, kita perlu mengirim data post ke view `welcome.blade.php` melalui `HomeController` . Post yang akan ditampilkan hanya post dengan status **published**.

> **Notice:** Buat post dengan status published atau update status post menjadi published secara manual melalui database.

```
<?php

namespace App\Http\Controllers;

// Kode lain

class HomeController extends Controller
{
  public function index()
  {
    $setting = Setting::first();
    $categories = Category::select(['id','name','slug'])->get();
    $posts =
Post::with(['user'])->where('status',StatusEnum::PUBLISHED)->paginate(2);
    return view('welcome',compact('setting','categories','posts'));
  }
}
```

> **Notice:** paginate pada post dapat disuaikan sendiri tidak harus 2.

Kemudian agar tampil di halaman homepage, ubah `welcome.blade.php` menjadi seperti ini

```
<x-frontend
  :setting="$setting"
  :categories="$categories"
  >
  @foreach ($posts as $post)
    <div class="post-preview">
      <a href="post.html">
        <h2 class="post-title">{{$post->title}}</h2>
        <h3 class="post-subtitle">{{$post->body}}</h3>
      </a>
      <p class="post-meta">
        Posted by
        <a href="#!">
          {{$post->user->name}}
        </a> on {{$post->published_at}}
      </p>
    </div>
  @endforeach
</x-frontend>
```

Hasilnya post akan muncul di halaman homepage

Laravel 10

Laravel versi 10 adalah versi selanjutnya untuk Laravel, dan ini telah direncanakan rilis pada 7 februari 2023. Dalam artikel ini, sebelum framework ini rilis versi terbarunya,

Posted by Toni Listiyo on 2023-02-09

CRUD Laravel

Laravel adalah framework PHP yang sangat populer saat ini dan memiliki banyak fitur yang sangat membantu dalam pengembangan aplikasi web. Salah satu fitur yang sangat penting dan sering digunakan dalam pengembangan aplikasi adalah CRUD. CRUD adalah singkatan dari Create, Read, Update, dan Delete, yang merupakan operasi dasar dalam pengembangan aplikasi. Laravel memudahkan proses pembuatan CRUD dengan menyediakan beberapa fitur seperti migrasi database, controller, dan model. Proses pembuatan CRUD menjadi sangat mudah dan efisien dengan menggunakan Laravel. Cukup dengan

Exceprt Post

Meskipun seluruh isi blog dapat dilihat dan terlihat lengkap di halaman, kita ingin menampilkan hanya beberapa bagian atau potongan yang memuat ringkasan dari isi blog agar lebih mudah dibaca dan tidak memenuhi seluruh ruang halaman.

Untuk melakukannya bisa memanfaatkan helper `Str::excerpt()` yang berguna untuk mempersingkat sebuah string, biasanya digunakan untuk menampilkan teks singkat dari sebuah konten yang lebih panjang dan menambahkan tiga titik (...) pada akhir potongan tersebut, seperti artikel atau posting blog.

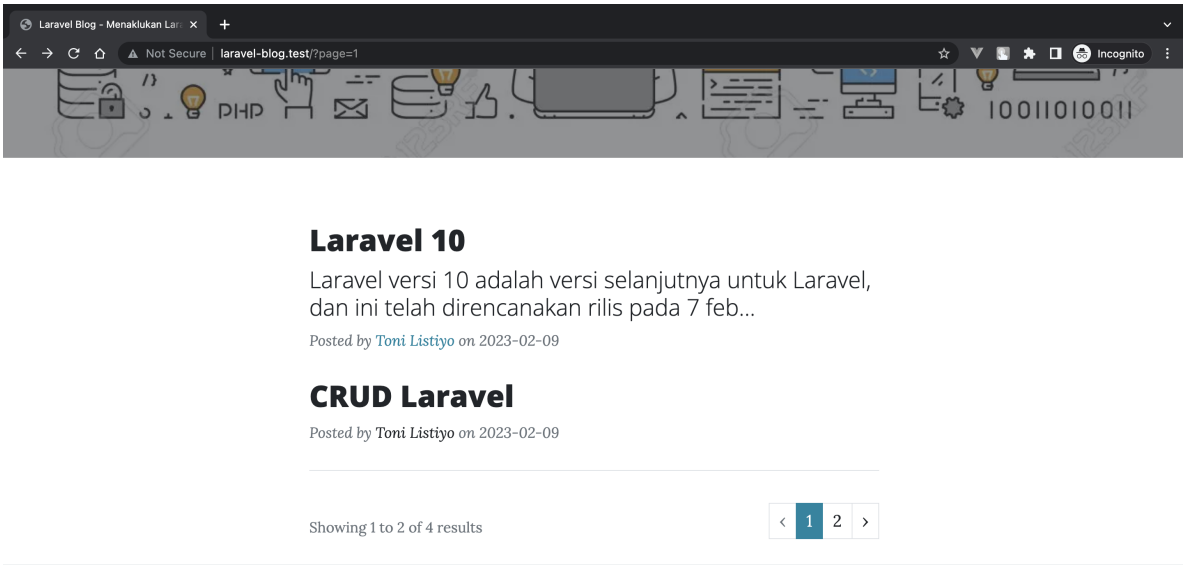
Tambahkan method `excerpt` di model `Post`

```php
<?php

namespace App\Models;

use Illuminate\Support\Str;


class Post extends Model
{
  // kode lain

  public function excerpt()
  {
    return Str::excerpt($this->body);
  }
}
```

Ubah pemanggilan `$post->body` menjadi `$post->excerpt()`

```
<x-frontend :setting="$setting" :categories="$categories">
 @foreach ($posts as $post)
 <div class="post-preview">
  <a href="post.html">
   <!-- Kode lain -->
   <h3 class="post-subtitle">{{$post->excerpt()}}</h3>
  </a>
  <!-- Kode lain -->
 </div>
 @endforeach

</x-frontend>
```

Tampilan blog sekarang menjadi lebih ringkas



Single Post

Untuk menampilkan setiap post secara detail dan lengkap, kita akan membuat tautan atau link yang akan mengarahkan pengguna ke halaman detail post. Halaman detail post akan menampilkan informasi yang terkait dengan post tersebut, dan post yang akan ditampilkan pada halaman tersebut akan dipilih berdasarkan slug (identifikasi unik yang diterapkan pada setiap post), bukan berdasarkan nomor identifikasi (ID) unik yang diterapkan pada setiap post.

Buat sebuah *invokable controller*, *invokable Controller* adalah jenis Controller di Laravel yang hanya memiliki satu method *public* yang dapat dipanggil langsung

```
php artisan make:controller ShowPostController -i
```

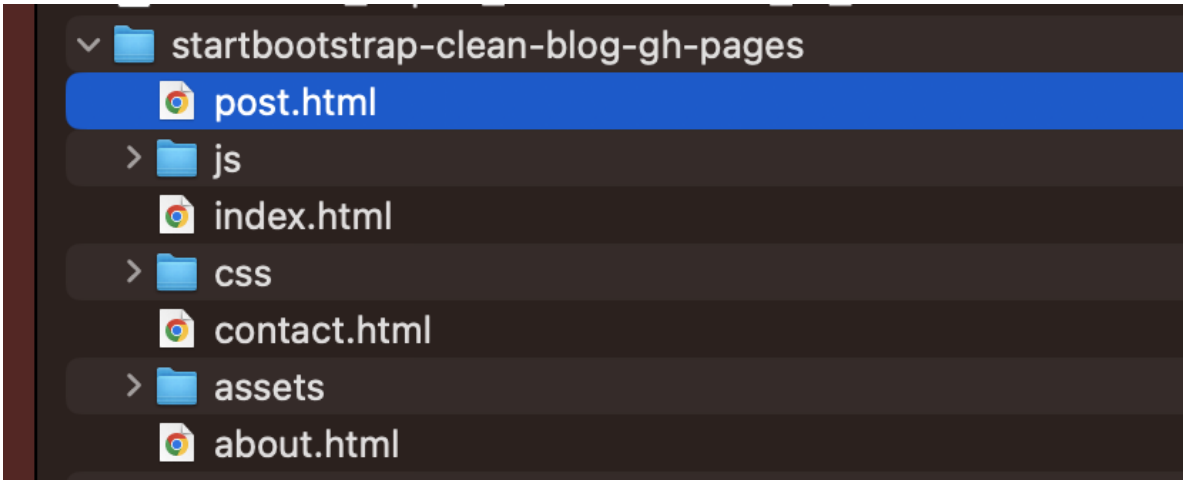
Di `ShowPostController` , selain mengirim data post, kita juga masih perlu mengirim data setting dan kategori karena layout frontend membutuhkan data setting dan kategori. Akan kita perbaiki nanti.

```
<?php

namespace App\Http\Controllers;

use App\Models\Post;
use App\Models\Setting;
use App\Models\Category;
use Illuminate\Http\Request;

class ShowPostController extends Controller
{
  public function __invoke(Request $request,string $slug)
  {
    $post = Post::whereSlug($slug)->first();
    $setting = Setting::first();
    $categories = Category::select(['id', 'name', 'slug'])->get();
    return view('post',compact('setting', 'categories','post'));
  }
}
```

Buat route untuk `ShowPostController`

```php
<?php

// Kode Lain

use App\Http\Controllers\ShowPostController;

Auth::routes();

Route::get('/', [App\Http\Controllers\HomeController::class,
'index'])->name('home');
Route::get('post/{slug}',ShowPostController::class)->name('show.post');

// Kode lain
```

Buat view dengan nama post.blade.php referensinya ada di file `post.html` dari template yang kita download



Jika kalian perhatikan, cover header dari single post berbeda dengan halaman home page, ini artinya header seharusnya dinamis, yang kita buat sebelumya adalah statis. Kita perlu ubah lagi layout untuk `frontend.blade.php`.

```
<!DOCTYPE html>
<html lang="en">
  @include('front.head')
  <body>
    <!-- Navigation-->
    @include('front.nav')
    <!-- Page Header-->
    {{$cover}}
    <!-- Main Content-->
    {{ $slot }}

    <!-- Footer-->
    @include('front.footer')
    <!-- Bootstrap core JS-->
    @include('front.scripts')
  </body>
</html>
```

Variabel `$cover` merupakan slot, jadi di `layout.blade.php` ada multiple slot. Cara penggunaan slot `$cover` dengan menggunakan tag `<x-slot:cover>` . Kita juga perlu mengubah `welcome.blade.php`

```
<x-frontend :setting="$setting" :categories="$categories">
  <x-slot:cover>
   @include('front.header')
  </x-slot:cover>
  <!-- Kode Lain -->
</x-frontend>
```

Full Source Code dapat dilihat di sini

Kembali lagi ke `post.blade.php` , kodenya seperti ini

```
<x-frontend :setting="$setting" :categories="$categories">
 <x-slot:cover>
  <!-- Source code Cover -->
 </x-slot:cover>
 <!-- Kode Lain -->
</x-frontend>
```

Full Source Code dapat dilihat di sini

Post Berdasarkan Kategori

Untuk menampilkan post berdasarkan kategori pertama buat sebuah invokable controller

```
php artisan make:controller ShowPostByCategory -i
```

Buat route untuk mengarahkan url ke `ShowPostByCategory`

```
Route::get('posts/category/{slug}',ShowPostByCategory::class)->name('posts.cat
egory');
```

Pada `nav.blade.php` , buat link di dropdown kategori menggunakan route yang telah kita buat

```
<nav class="navbar navbar-expand-lg navbar-light" id="mainNav">

 <!-- Kode lain -->

 <ul class="dropdown-menu border border-0">
 @foreach ($categories as $category)
  <li>
    <a
      class="dropdown-item"
      href="{{route('posts.category',$category->slug)}}">
      {{$category->name}}
    </a>
 </li>
 @endforeach
</ul>

 <!-- Kode lain -->
</nav>
```

Pada `ShowPostByCategory` kita akan menggunakan view yang sama dengan halaman homepage. Kodenya seperti ini

```php
<?php

namespace App\Http\Controllers;

// Kode lain

class ShowPostByCategory extends Controller
{

  public function __invoke(Request $request,string $slug)
  {
    $category = Category::whereSlug($slug)->first();
    $posts = Post::with(['user'])
                ->whereBelongsTo($category)
                ->whereStatus(StatusEnum::PUBLISHED)
                ->paginate(2);
    $setting = Setting::first();
    $categories = Category::select(['id', 'name', 'slug'])->get();
    return view('welcome', compact('setting', 'categories', 'posts'));
  }
}
```

Penjelasan kode diatas sebagai berikut:

`Post::with(['user'])`: Memuat relasi user untuk model Post. Ini akan mengambil data user untuk setiap post yang diambil dengan menggunakan eager loading, sehingga menghindari masalah N+1 query.

`->whereBelongsTo($category)`: Menerapkan kondisi *where* pada model Post yang menunjukkan bahwa Post tersebut termasuk dalam kategori tertentu yang diidentifikasi dengan variabel `$category`.

`->whereStatus(StatusEnum::PUBLISHED)`: Menerapkan kondisi *where* pada model Post yang menunjukkan bahwa Post tersebut memiliki status *published*, yang diidentifikasi dengan konstanta PUBLISHED yang didefinisikan dalam sebuah enum bernama StatusEnum.

Jika kita klik navbar dropdown kategori, akan muncul post berdasarkan kategori yang dipilih.

Publish Post

Fitur ini digunakan untuk mempublish post yang statusnya masih draft, fitur ini berupa tombol aksi yang ketika diklik akan merubah status post dari draft menjadi published.

Buat controller untuk menangani publish post

```
php artisan make:controller Admin/PublishPostController -i
```

Buat route untuk mengarahkan ke controller tersebut. Pada dasarnya publish post adalah update post merubah status dari draft menjadi published jadi method yang digunakan adalah `PUT`.

```
Route::group(['prefix'=>'admin','middleware'=>'auth'],function(){
  // Kode lain
Route::put('posts/{post}/publish',PublishPostController::class)->name('posts.p
ublish');
});
```

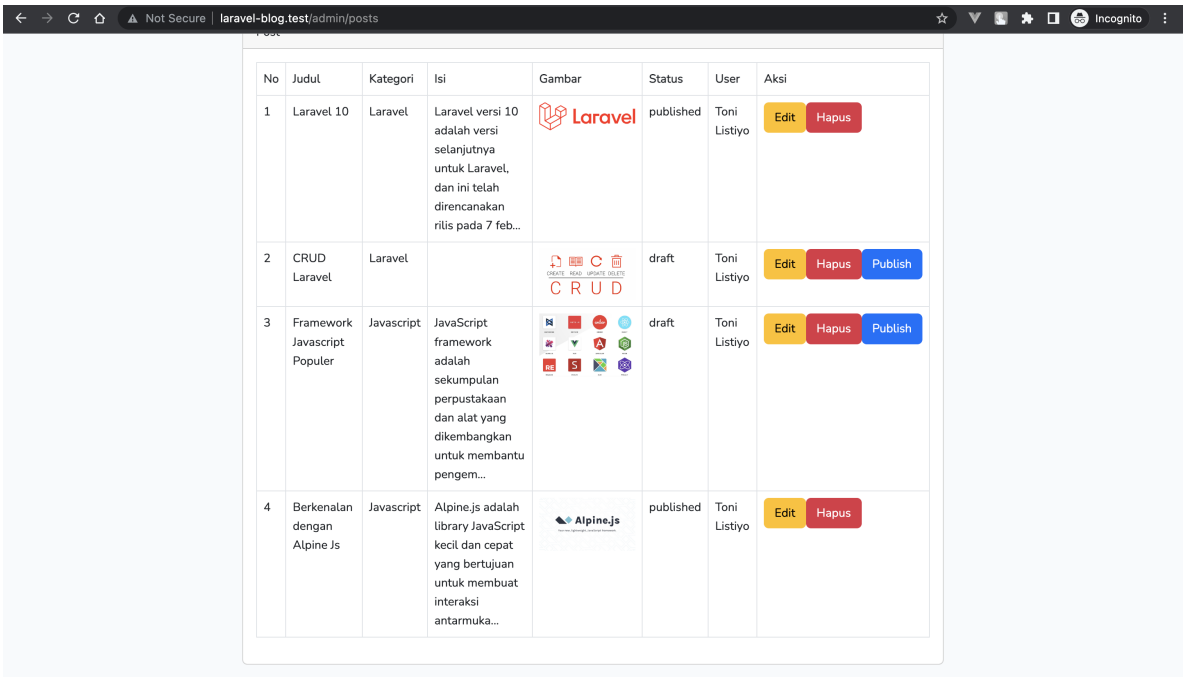
Pada `StatusEnum` buat method untuk mengecek apakah status post `PUBLISHED` . Method ini akan kita gunakan di view index untuk menampilkan tombol publish jika status post bukan published.

```php
<?php

namespace App\Enums;

enum StatusEnum:string
{
  case DRAFT = 'draft';
  case PUBLISHED = 'published';
  case ARCHIVED = 'archived';

  public function isPublished()
  {
    return $this == self::PUBLISHED;
  }

}
```

Di file `posts/index.blade.php` tambahkan tombol untuk publish di kolom aksi. Tombol tersebut akan muncul jika status bukan published. Ubah juga pemanggilan `$post->body` menjadi `$post->excerpt()` agar tidak seluruh isi post tampil.

```
@if (!$post->status->isPublished())
  <form action="{{route('posts.publish',$post)}}" method="POST">
    @csrf
    @method('PUT')
    <button onclick="return confirm('Apakah anda yakin akan publish post?')"
class="btn btn-primary"
        type="submit">Publish</button>
  </form>
@endif
```

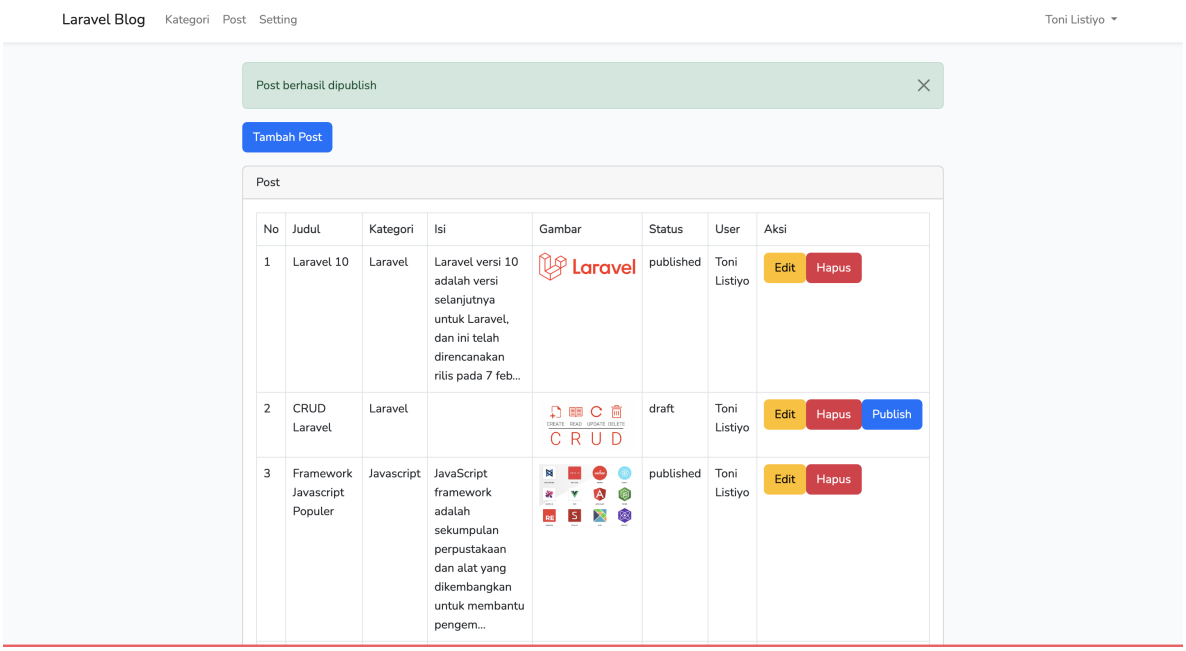
Tampilan halaman index post akan seperti ini, tombol published hanya akan muncul jika status bukan `published`.



Sedangkan kode untuk update status menjadi published seperti ini

```php
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PublishPostController extends Controller
{
  public function __invoke(Request $request,Post $post)
  {
    $post->update([
      'status'=>StatusEnum::PUBLISHED,
      'published_at'=>now()
    ]);
    return redirect()->route('posts.index')->with('success', 'Post berhasil dipublish');

  }
}
```

Berikut ini hasilnya ketika publish post melalui tombol publish


Laravel Blog
Kategori
Post
Setting
Toni Listiyo
Post berhasil dipublish
Tambah Post
Post
No
Judul
Kategori
Isi
Gambar
Status
User
Aksi
1
Laravel 10
Laravel
Laravel versi 10 adalah versi selanjutnya untuk Laravel, dan ini telah direncanakan rilis pada 7 feb...
published
Toni Listiyo
Edit
Hapus
2
CRUD Laravel
Laravel
draft
Toni Listiyo
Edit
Hapus
Publish
3
Framework Javascript Populer
Javascript
JavaScript framework adalah sekumpulan perpustakaan dan alat yang dikembangkan untuk membantu pengem...
published
Toni Listiyo
Edit
Hapus

Badge Status

Badge status digunakan untuk memberi warna status post, tujuannya adalah untuk memberikan representasi visual dan cepat dari jenis status, sehingga hanya dengan melihat warna saja kita sudah tau status dari blog.

Saat ini status untuk post ada 3 yaitu: draft, published, dan archieved. Kita akan membuat warna untuk masing - masing status menggunakan Badge Bootstrap.

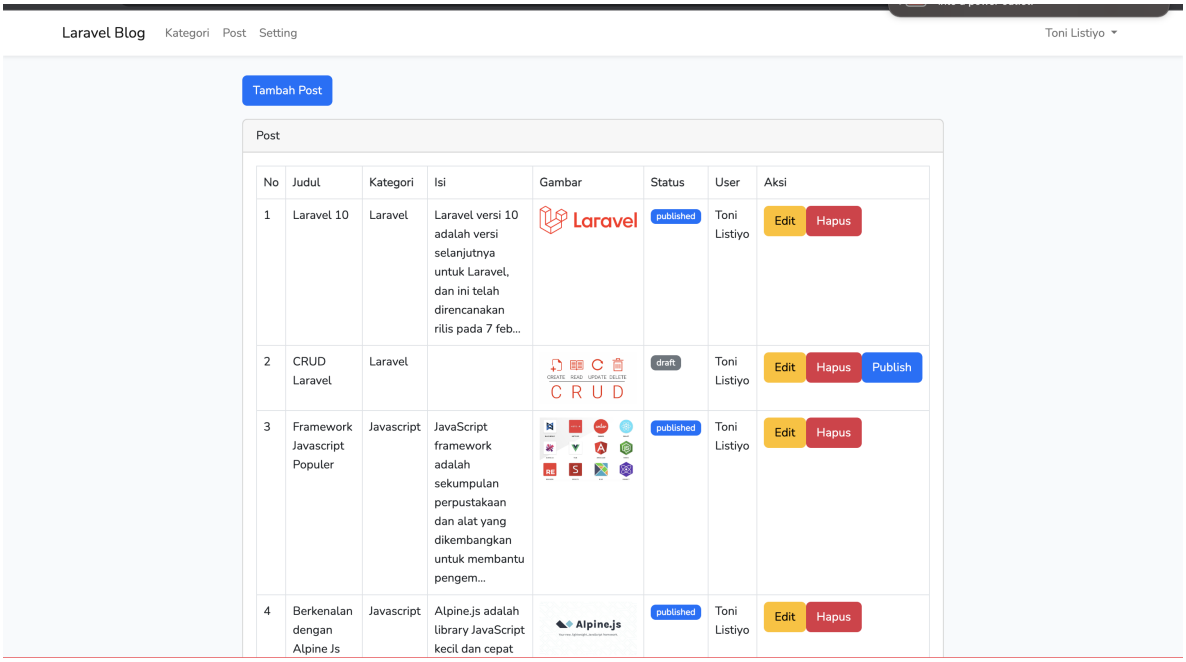
Buat method baru di `StatusEnum` model dengan nama `color()`

```
<?php

namespace App\Enums;

enum StatusEnum:string
{

  //  Kode lain

  public function color()
  {
    return match($this){
      self::DRAFT=>'secondary',
      self::PUBLISHED=>'primary',
      self::ARCHIVED=>'danger'
    };
  }
}
```

Pada file `posts/index.blade.php` di kolom status ubah kodenya menjadi seperti ini

```
<span class="badge text-bg-
{{$post->status->color()}}">{{$post->status}}</span>
```

Tampilan status sudah berubah menjadi berwarna


Laravel Blog
Kategori
Post
Setting
Toni Listiyo
Tambah Post
Post
No
Judul
Kategori
Isi
Gambar
Status
User
Aksi
1
Laravel 10
Laravel
Laravel versi 10 adalah versi selanjutnya untuk Laravel, dan ini telah direncanakan rilis pada 7 feb...
published
Toni Listiyo
Edit
Hapus
2
CRUD Laravel
Laravel
draft
Toni Listiyo
Edit
Hapus
Publish
3
Framework Javascript Populer
Javascript
JavaScript framework adalah sekumpulan perpustakaan dan alat yang dikembangkan untuk membantu pengem...
published
Toni Listiyo
Edit
Hapus
4
Berkenalan dengan Alpine Js
Javascript
Alpine.js adalah library JavaScript kecil dan cepat
published
Toni Listiyo
Edit
Hapus

Save and Publish Post

Saat membuat post secara default statusnya adalah *draft* , kita akan menambahkan pilihan apakah user ingin menyimpan sebagai *draft* atau langsung *publish*.

Tambahkan dropdown status di `create.blade.php` dan `edit.blade.php` . Nilai dropdown berasal dari `StatusEnum` yang dikirim melalui `PostController`. Kemudian tambahkan data status di `PostController` yang akan dikirim ke view .

```
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PostController extends Controller
{

  //kode lain

  public function create()
  {
    $categories = Category::all();
    $statuses = array_filter($status =
StatusEnum::cases(),fn($status)=>$status->value !=
StatusEnum::ARCHIVED->value);
    return view('admin.posts.create',compact('categories','statuses'));
  }

  // Kode lain
    public function edit(Post $post)
    {
      $categories = Category::all();
      $statuses = array_filter($status = StatusEnum::cases(), fn ($status) =>
$status->value != StatusEnum::ARCHIVED->value);
      return view('admin.posts.edit',compact('post','categories','statuses'));
    }

  // Kode lain
}
```

Penjelasan nilai variabel `$statuses` adalah sebagai berikut :

1. `StatusEnum::cases()`: Mengambil semua nilai yang didefinisikan pada class enum

`StatusEnum`, yaitu **DRAFT**, **PUBLISHED**, dan **ARCHIVED**. Method cases() merupakan method bawaan yang terdapat pada class enum di PHP 8.0.

2. `array_filter()`: Menerapkan filter pada setiap elemen array yang diberikan, dan mengembalikan array yang hanya berisi elemen yang memenuhi kriteria filter tersebut. Pada kode di atas, parameter pertama yang diberikan adalah $status = `StatusEnum::cases()` yaitu variabel yang berisi semua nilai yang didefinisikan pada class enum `StatusEnum`. Parameter kedua yang diberikan adalah callback function (fungsi yang diberikan sebagai argumen) yang menentukan kriteria filternya.

3. `fn($status)=>$status->value != StatusEnum::ARCHIVED->value`: Merupakan sebuah anonymous function (fungsi tanpa nama) yang digunakan sebagai callback function pada array_filter(). Fungsi ini mengambil setiap elemen `$status` dari array yang diberikan, dan membandingkan nilai properti value dari `$status` dengan nilai properti value dari konstan **ARCHIVED** pada class enum `StatusEnum`. Jika nilai properti value dari `$status` tidak sama dengan nilai properti value dari konstan **ARCHIVED**, maka fungsi ini akan mengembalikan true sehingga `$status` akan disimpan pada array hasil filter.

Kesimpulannya adalah kita hanya mengambil status selain **ARCHIEVED** yang akan dikirim ke `create.blade.php` dan `edit,blade.php`.

Tambahkan dropdown status di `create.blade.php` setelah tag `<textarea>`

```
{{-- Kode lain --}}
<div class="mb-3">
   <label for="status" class="form-label">Status</label>
   <select name="status" class="form-control @error('status') is-invalid
@enderror" id="status">
     <option value="">--Pilih Status--</option>
     @foreach ($statuses as $status)
     <option {{old('status')==$status->value ?'selected':''}}
         value="{{$status->value}}">{{$status->name}}</option>
     @endforeach
   </select>
   @error('status')
   <div class="invalid-feedback">
     {{$message}}
   </div>
   @enderror
</div>
{{-- Kode lain --}}
```

Dibawah ini untuk `edit.blade.php`

```
{{-- Kode lain --}}
<div class="mb-3">
  <label for="status" class="form-label">Status</label>
  <select name="status" class="form-control @error('status') is-invalid
@enderror" id="status">
     <option value="">--Pilih Status--</option>
     @foreach ($statuses as $status)
     <option {{$post->status == $status->value ?'selected':''}}
         value="{{$status->value}}">{{$status->name}}</option>
     @endforeach
  </select>
  @error('status')
  <div class="invalid-feedback">
    {{$message}}
  </div>
 @enderror
</div>
{{-- Kode lain --}}
```

Kemudian di method `store` dan `update` , tambahkan nilai status dari form ke dalam data yang akan disimpan atau diupdate.

```php
<?php

namespace App\Http\Controllers\Admin;

// Kode lain

class PostController extends Controller
{
  // Kode lain
  public function store(Request $request)
  {
    $request->validate([
    //kode lain
      'status'=>'required'
    ]);
    Post::create([
      // Kode lain
      'status'=>$request->status,
      'published_at'=>$request->status==StatusEnum::PUBLISHED->value ?
now():null,
      // Kode lain
    ]);

  }

  public function update(Request $request, Post $post)
  {
    $request->validate([
      //kode lain
      'status'=>'required'
    ]);
    $post->update([
      // Kode lain
      'status' => $request->status,
      'published_at' => $request->status == StatusEnum::PUBLISHED->value ?
now() : null,
      // Kode kain
     ]);
  }

  // Kode lain
}
```

Demikianlah study kasus membuat blog sederhana menggunakan laravel 9 dan

bootstrap 5. Terima kasih telah membaca panduan ini, semoga bermanfaat bagi Anda yang ingin mempelajari cara membuat blog sederhana menggunakan Laravel 9. Selamat mencoba dan semoga sukses!

Jika kalian berminat belajar lebih lanjut kalian dapat membeli versi fullnya. Link ada di halaman terakhir.