Materi Pelatihan Laravel

Penulis: Manus AI
Tanggal: 27 Juli 2025

Daftar Isi

22. Laravel Eloquent
23. Model
24. Relationship
25. Model, View, Controller Workflow
26. SQLite
27. Make Simple Project in Laravel
28. CRUD Application
29. Authentication and Authorization

Pendahuluan

Laravel adalah salah satu framework PHP yang paling populer dan powerful untuk pengembangan aplikasi web modern. Framework ini dikembangkan oleh Taylor Otwell pada tahun 2011 dan telah menjadi pilihan utama bagi developer PHP di seluruh dunia. Laravel menggunakan arsitektur Model-View-Controller (MVC) yang memisahkan logika bisnis, presentasi, dan data dengan jelas.

Materi pelatihan ini dirancang untuk memberikan pemahaman komprehensif tentang Laravel mulai dari konsep dasar hingga implementasi aplikasi yang kompleks. Setiap topik akan disertai dengan penjelasan teoritis yang mendalam, contoh kode praktis, dan best practices yang direkomendasikan oleh komunitas Laravel.

Laravel menawarkan berbagai fitur canggih seperti Eloquent ORM untuk manajemen database, Blade templating engine untuk view, Artisan CLI untuk otomatisasi tugas, dan sistem routing yang fleksibel. Framework ini juga dilengkapi dengan tools untuk testing, caching, session management, dan authentication yang memudahkan developer dalam membangun aplikasi web yang robust dan scalable.

Dalam pelatihan ini, peserta akan mempelajari cara menggunakan Laravel untuk membangun aplikasi web dari awal, mulai dari setup environment, pembuatan route dan controller, pengelolaan database dengan migration dan seeder, hingga implementasi fitur authentication dan authorization. Setiap konsep akan dijelaskan dengan detail dan disertai contoh implementasi yang dapat langsung dipraktikkan.

1. Perkenalan Laravel

Laravel adalah sebuah *framework* aplikasi web PHP *open-source* yang dirancang untuk pengembangan aplikasi web dengan pola arsitektur Model-View-Controller (MVC). Dibuat oleh Taylor Otwell, Laravel dikenal karena sintaksnya yang elegan, ekspresif, dan mudah dipahami, yang bertujuan untuk membuat proses pengembangan menjadi lebih menyenangkan dan produktif bagi para *developer*.

Mengapa Menggunakan Laravel?

Ada beberapa alasan utama mengapa Laravel menjadi pilihan populer di kalangan *developer*:

- **Sintaks yang Elegan dan Ekspresif:** Laravel menyediakan sintaks yang bersih dan mudah dibaca, yang memungkinkan *developer* menulis kode dengan lebih cepat dan efisien.
- **Fitur Lengkap:** Laravel dilengkapi dengan berbagai fitur *built-in* yang sangat membantu dalam pengembangan aplikasi web, seperti sistem *routing* yang kuat, *ORM* (Object-Relational Mapper) Eloquent untuk interaksi *database*, *templating engine* Blade, sistem *authentication* dan *authorization*, serta *queueing* dan *caching*.
- **Ekosistem yang Kaya:** Laravel memiliki ekosistem yang luas dengan banyak *package* dan *tool* pihak ketiga yang dapat diintegrasikan dengan mudah. Komunitasnya juga sangat aktif dan suportif.
- **Performa:** Dengan fitur-fitur seperti *caching* dan *queueing*, Laravel dapat membantu *developer* membangun aplikasi yang memiliki performa tinggi.
- **Keamanan:** Laravel menyediakan berbagai fitur keamanan *built-in* untuk melindungi aplikasi dari serangan umum seperti *SQL injection*, *Cross-Site Scripting (XSS)*, dan *Cross-Site Request Forgery (CSRF)*.
- **Skalabilitas:** Laravel dirancang untuk dapat diskalakan, sehingga cocok untuk membangun aplikasi dari skala kecil hingga besar.

Arsitektur MVC dalam Laravel

Laravel mengadopsi pola arsitektur Model-View-Controller (MVC), yang memisahkan logika aplikasi menjadi tiga komponen utama:

- **Model:** Bertanggung jawab untuk mengelola data dan logika bisnis yang terkait dengan data tersebut. Model berinteraksi langsung dengan *database*.
- **View:** Bertanggung jawab untuk menampilkan data kepada pengguna. Dalam Laravel, *view* biasanya dibuat menggunakan *templating engine* Blade.
- **Controller:** Bertindak sebagai perantara antara Model dan View. *Controller* menerima *request* dari pengguna, memprosesnya (seringkali dengan berinteraksi dengan Model), dan kemudian mengembalikan *response* (biasanya berupa *view*).

Pemisahan ini membuat kode lebih terorganisir, mudah dipelihara, dan memungkinkan *developer* untuk bekerja secara kolaboratif pada bagian-bagian yang berbeda dari aplikasi tanpa saling mengganggu.

Contoh Sederhana: Hello World di Laravel

Untuk memberikan gambaran awal, mari kita lihat bagaimana membuat *response*

sederhana 'Hello World' di Laravel. Ini melibatkan definisi *route* yang akan mengembalikan teks 'Hello World'.

```
// file: routes/web.php

use Illuminate\Support\Facades\Route;

Route::get('/hello', function () {
    return 'Hello World from Laravel!';
});
```

Dalam contoh di atas:

- `Route::get('/hello', ...)`: Mendefinisikan *route* yang akan merespons *request* HTTP GET ke URL `/hello`.
- `function () { ... }`: Ini adalah *closure* (fungsi anonim) yang akan dieksekusi ketika *route* `/hello` diakses.
- `return 'Hello World from Laravel!';`: Mengembalikan string 'Hello World from Laravel!' sebagai *response* HTTP.

Ketika Anda mengakses `http://your-laravel-app.test/hello` di *browser*, Anda akan melihat teks 'Hello World from Laravel!' ditampilkan. Ini adalah dasar bagaimana Laravel menangani *request* dan *response*.

2. Perkenalan Fitur Laravel

Laravel dikenal dengan kekayaan fiturnya yang dirancang untuk mempercepat dan mempermudah pengembangan aplikasi web. Berikut adalah beberapa fitur utama yang membuat Laravel menjadi *framework* yang sangat populer:

2.1. Artisan Console

Artisan adalah *command-line interface (CLI)* yang disertakan dengan Laravel. Artisan menyediakan sejumlah perintah yang membantu *developer* dalam tugas-tugas pengembangan sehari-hari, seperti membuat *controller*, *model*, *migration*, *seeder*, menjalankan *testing*, dan banyak lagi. Ini sangat meningkatkan produktivitas *developer*.

Contoh Penggunaan Artisan:

Membuat *controller* baru:

```
php artisan make:controller PhotoController
```

Menjalankan *migration database*:

```
php artisan migrate
```

2.2. Eloquent ORM

Eloquent adalah *Object-Relational Mapper (ORM)* yang kuat dan elegan yang disertakan dengan Laravel. Eloquent menyediakan cara yang indah dan ekspresif untuk berinteraksi dengan *database* Anda. Setiap tabel *database* memiliki

Model yang sesuai, yang digunakan untuk berinteraksi dengan tabel tersebut.

Contoh Penggunaan Eloquent:

```php
// Mengambil semua user
$users = App\Models\User::all();

// Mencari user berdasarkan ID
$user = App\Models\User::find(1);

// Membuat user baru
$user = new App\Models\User;
$user->name = 'John Doe';
$user->email = 'john@example.com';
$user->password = bcrypt('password');
$user->save();
```

2.3. Blade Templating Engine

Blade adalah *templating engine* yang sederhana namun kuat yang disediakan oleh Laravel. Blade memungkinkan Anda menggunakan sintaks PHP biasa dalam *view* Anda, tetapi juga menyediakan *shortcut* yang nyaman untuk tugas-tugas umum seperti menampilkan data, *looping*, dan *conditional statement*. File *view* Blade memiliki ekstensi `.blade.php`.

Contoh Penggunaan Blade:

```blade
<!-- file: resources/views/welcome.blade.php -->

<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Hello, {{ $name }}!</h1>

    @if (count($users) > 0)
        <ul>
            @foreach ($`users as `$user)
                <li>{{ $user->name }}</li>
            @endforeach
        </ul>
    @else
        <p>No users found.</p>
    @endif
</body>
</html>
```

2.4. Routing

Laravel menyediakan sistem *routing* yang fleksibel dan ekspresif yang memungkinkan Anda mendefinisikan bagaimana aplikasi Anda merespons *request* HTTP. Anda dapat

mendefinisikan *route* untuk berbagai metode HTTP (GET, POST, PUT, DELETE) dan mengaitkannya dengan *closure* atau *controller action*.

Contoh Penggunaan Routing:

```
// file: routes/web.php

use Illuminate\Support\Facades\Route;

Route::get(
    '/users/{id}',
    function ($id) {
        return 'User ID: ' . $id;
    }
);

Route::post(
    '/submit-form',
    function () {
        // Handle form submission
    }
);
```

2.5. Middleware

Middleware menyediakan mekanisme yang nyaman untuk memfilter *request* HTTP yang masuk ke aplikasi Anda. Misalnya, Laravel menyertakan *middleware* untuk memverifikasi apakah pengguna diautentikasi, *middleware* CORS, dan banyak lagi. Anda juga dapat membuat *middleware* kustom Anda sendiri.

Contoh Penggunaan Middleware:

```
// file: routes/web.php

Route::get(
    '/admin/dashboard',
    function () {
        // Only authenticated users can access this
    }
)->middleware('auth');
```

2.6. Authentication dan Authorization

Laravel menyediakan solusi *built-in* yang kuat untuk *authentication* (memverifikasi identitas pengguna) dan *authorization* (memverifikasi apakah pengguna memiliki izin untuk melakukan tindakan tertentu). Ini mencakup sistem *scaffolding* yang cepat untuk *login*, *registration*, *password reset*, dan *email verification*.

Contoh Penggunaan Authentication:

```
 // Mengecek apakah user sudah login
if (Auth::check()) {
    // User is logged in...
}

// Mengambil user yang sedang login
$user = Auth::user();
```

---

3. Instalasi Laravel

Untuk memulai pengembangan dengan Laravel, Anda perlu menginstal beberapa *software* dan mengikuti langkah-langkah instalasi. Laravel memiliki persyaratan sistem tertentu yang harus dipenuhi.

3.1. Persyaratan Sistem

Sebelum menginstal Laravel, pastikan server Anda memenuhi persyaratan berikut:

- PHP >= 8.2
- Ekstensi PHP: Ctype, cURL, DOM, Fileinfo, Mbstring, OpenSSL, PCRE, PDO, Session, XML, Zip
- Composer (Manajer dependensi PHP)

Sebagian besar *web server* modern seperti Apache atau Nginx dengan PHP sudah terinstal akan memenuhi persyaratan ini. Jika Anda menggunakan lingkungan pengembangan lokal seperti XAMPP, WAMP, atau Laragon, biasanya semua ekstensi PHP yang diperlukan sudah aktif secara *default*.

3.2. Menginstal Composer

Composer adalah *tool* manajemen dependensi untuk PHP. Ini memungkinkan Anda mendeklarasikan *library* yang dibutuhkan proyek Anda, dan Composer akan menginstal serta mengelolanya untuk Anda. Jika Anda belum memiliki Composer, Anda dapat menginstalnya dengan mengikuti instruksi di situs web resmi Composer: https://getcomposer.org/download/

Pastikan Composer terinstal secara global sehingga Anda dapat menjalankannya dari *command line* di direktori mana pun.

3.3. Menginstal Laravel

Ada dua cara utama untuk menginstal Laravel:

a. Melalui Composer Create-Project

Ini adalah cara yang paling umum dan direkomendasikan untuk memulai proyek Laravel baru. Perintah ini akan mengunduh *framework* Laravel dan semua dependensinya, lalu membuat proyek baru di direktori yang Anda tentukan.

```
composer create-project laravel/laravel example-app
```

- `laravel/laravel` : Ini adalah *package* Laravel yang akan diunduh.
- `example-app` : Ini adalah nama direktori tempat proyek Laravel Anda akan dibuat. Anda bisa menggantinya dengan nama proyek Anda sendiri.

Setelah perintah ini selesai, Anda akan memiliki instalasi Laravel yang bersih di dalam direktori `example-app` .

b. Melalui Laravel Installer

Laravel Installer adalah cara cepat untuk membuat proyek Laravel baru setelah Anda menginstalnya secara global. Pertama, instal Laravel Installer menggunakan Composer:

```
composer global require laravel/installer
```

Pastikan direktori `~/.composer/vendor/bin` (macOS/Linux) atau `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin` (Windows) ada di variabel lingkungan PATH sistem Anda, sehingga *executable* `laravel` dapat ditemukan.

Setelah Laravel Installer terinstal, Anda dapat membuat proyek baru dengan perintah:

```
laravel new example-app
```

Perintah ini akan melakukan hal yang sama dengan `composer create-project`, tetapi dengan sintaks yang lebih ringkas.

3.4. Menjalankan Aplikasi Laravel

Setelah instalasi selesai, navigasikan ke direktori proyek Anda:

```
cd example-app
```

Kemudian, Anda dapat menjalankan *development server* Laravel menggunakan Artisan:

```
php artisan serve
```

Perintah ini akan memulai *server* pengembangan PHP di `http://127.0.0.1:8000` (atau port lain yang tersedia). Anda dapat membuka URL ini di *browser* Anda, dan Anda akan melihat halaman selamat datang Laravel.

Jika Anda menggunakan Docker, Anda bisa menggunakan Laravel Sail, yang menyediakan lingkungan pengembangan Docker yang ringan untuk membangun aplikasi Laravel. Anda dapat menginstal Sail dengan menambahkan flag `--with-sail` saat membuat proyek baru:

```
composer create-project laravel/laravel example-app --with-sail
```

Setelah itu, Anda dapat menjalankan aplikasi Anda dengan `vendor/bin/sail up`.

---

4. Laravel Route

Routing adalah salah satu fitur inti dalam Laravel yang memungkinkan Anda mendefinisikan bagaimana aplikasi Anda merespons *request* HTTP yang masuk. Laravel menyediakan cara yang sangat ekspresif dan fleksibel untuk mendefinisikan *route*.

Semua *route* Laravel didefinisikan dalam file *route* yang terletak di direktori `routes`. Ada beberapa file *route* yang berbeda, masing-masing dengan tujuan yang berbeda:

- `routes/web.php` : Berisi *route* untuk antarmuka web Anda. *Route* ini biasanya memiliki *state* sesi dan perlindungan CSRF.
- `routes/api.php` : Berisi *route* untuk API Anda. *Route* ini tidak memiliki *state* sesi dan dimaksudkan untuk menjadi *stateless*.
- `routes/console.php` : Berisi definisi *closure* berbasis konsol.
- `routes/channels.php` : Berisi semua pendaftaran *broadcast channel* yang didukung aplikasi Anda.

Dalam materi ini, kita akan fokus pada `routes/web.php`.

4.1. Basic Routing

Route paling dasar di Laravel terdiri dari metode HTTP dan URI yang akan merespons, serta *closure* atau *controller action* yang akan dieksekusi ketika *route* tersebut cocok.

Contoh:

```
// file: routes/web.php

use Illuminate\Support\Facades\Route;

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

Route::post(
    '/register',
    function () {
        // Handle user registration
    }
);

Route::put(
    '/users/{id}',
    function ($id) {
        // Update user with ID $id
    }
);

Route::delete(
    '/users/{id}',
    function ($id) {
        // Delete user with ID $id
    }
    );

// Route yang merespons berbagai metode HTTP
Route::match(['get', 'post'], '/multi-method', function () {
    return 'This route responds to both GET and POST requests.';
});

// Route yang merespons semua metode HTTP
Route::any('/any-method', function () {
    return 'This route responds to any HTTP verb.';
});
```

4.2. Route Parameter

Seringkali Anda perlu menangkap segmen URI dalam *route* Anda. Misalnya, Anda mungkin perlu menangkap ID pengguna dari URL. Anda dapat melakukannya dengan mendefinisikan parameter *route*.

Contoh:

```
// file: routes/web.php

Route::get(
    '/users/{id}',
    function ($id) {
        return 'User ID: ' . $id;
    }
);

Route::get(
    '/posts/{post}/comments/{comment}',
    function ($`postId, `$commentId) {
        return 'Post ID: ' . $`postId . ', Comment ID: ' . `$commentId;
    }
);
```

Anda juga dapat membuat parameter *route* menjadi opsional dengan menambahkan tanda `?` setelah nama parameter dan memberikan nilai *default* pada *closure* atau *controller action*.

Contoh Route Parameter Opsional:

```
// file: routes/web.php

Route::get(
    '/name/{name?}',
    function ($name = null) {
        return $`name ? 'Hello ' . `$name : 'Hello Guest';
    }
);
```

Regular Expression Constraints

Anda dapat membatasi format parameter *route* menggunakan metode `where()` pada instance *route*. Metode `where()` menerima nama parameter dan ekspresi reguler yang mendefinisikan bagaimana parameter tersebut harus dibatasi.

Contoh:

```
// file: routes/web.php

Route::get(
    '/user/{id}',
    function ($id) {
        return 'User ID: ' . $id;
    }
)->where('id', '[0-9]+'); // Hanya menerima angka

Route::get(
    '/product/{name}',
    function ($name) {
        return 'Product Name: ' . $name;
    }
)->where('name', '[A-Za-z]+'); // Hanya menerima huruf

Route::get(
    '/item/{id}/{name}',
    function ($`id, `$name) {
        return 'Item ID: ' . $`id . ', Name: ' . `$name;
    }
)->where(['id' => '[0-9]+', 'name' => '[A-Za-z]+']);
```

4.3. Named Route

Named route memungkinkan Anda untuk dengan mudah menghasilkan URL atau melakukan *redirect* untuk *route* tertentu. Anda dapat menentukan nama untuk *route* dengan merangkai metode `name()` ke definisi *route*.

Contoh:

```
// file: routes/web.php

Route::get(
    '/user/profile',
    function () {
        //
    }
)->name('profile');

// Menggunakan named route untuk menghasilkan URL
$url = route('profile');

// Menggunakan named route untuk redirect
return redirect()->route('profile');
```

Named route sangat berguna ketika Anda perlu mengubah URI dari *route* tertentu. Jika Anda telah menamai *route* Anda, Anda dapat memperbarui URI di file *route* Anda tanpa perlu memperbarui setiap instance di mana Anda telah menghasilkan URL ke *route* tersebut.

Named Route dengan Parameter

Jika *named route* Anda memiliki parameter, Anda dapat meneruskan parameter tersebut sebagai argumen kedua ke fungsi `route()`.

Contoh:

```
// file: routes/web.php

Route::get(
    '/user/{id}/profile',
    function ($id) {
        //
    }
)->name('user.profile');

$url = route('user.profile', ['id' => 1]); // Menghasilkan: /user/1/profile
```

4.4. Route Prefix

Anda dapat mengelompokkan *route* yang memiliki awalan URI yang sama menggunakan metode `prefix()`. Ini sangat berguna untuk mengelola *route* yang terkait dengan area tertentu dari aplikasi Anda, seperti panel admin.

Contoh:

```
// file: routes/web.php

Route::prefix('admin')->group(function () {
    Route::get(
        '/users',
        function () {
            return 'Admin Users Page';
        }
    );
    Route::get(
        '/products',
        function () {
            return 'Admin Products Page';
        }
    );
});
```

Dalam contoh di atas, ketika Anda mengakses `/admin/users`, *route* `admin/users` akan dieksekusi. Demikian pula, `/admin/products` akan mengeksekusi *route* `admin/products`.

Route Group dengan Middleware

Anda juga dapat menggabungkan `prefix()` dengan `middleware()` untuk menerapkan *middleware* ke sekelompok *route*.

Contoh:

```
 // file: routes/web.php

Route::middleware(['auth', 'admin'])->prefix('admin')->group(function () {
    Route::get(
        '/dashboard',
        function () {
            return 'Admin Dashboard';
        }
    );
    Route::get(
        '/settings',
        function () {
            return 'Admin Settings';
        }
    );
});
```

Ini akan memastikan bahwa semua *route* di dalam grup `admin` hanya dapat diakses oleh pengguna yang sudah diautentikasi dan memiliki peran admin.

---

5. Laravel Controller

Controller adalah bagian penting dari arsitektur MVC di Laravel. *Controller* bertanggung jawab untuk menangani logika *request* HTTP dan mengembalikan *response*. Daripada mendefinisikan semua logika penanganan *request* Anda sebagai *closure* di file *route*, Anda dapat mengaturnya menggunakan kelas *controller*.

Controller disimpan di direktori `app/Http/Controllers`. Anda dapat membuat *controller* baru menggunakan perintah Artisan `make:controller`.

5.1. Basic Controller

Controller dasar adalah kelas PHP sederhana yang meng-extend kelas `Controller` dasar Laravel. Kelas ini dapat berisi beberapa metode publik, yang masing-masing dapat dihubungkan ke *route*.

Membuat Basic Controller:

```
php artisan make:controller UserController
```

Ini akan membuat file baru di `app/Http/Controllers/UserController.php`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    //
}
```

Menambahkan Metode ke Controller:

Sekarang, mari tambahkan metode publik ke *controller* ini:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view(
            'user.profile',
            ['user' => User::findOrFail($id)]
        );
    }
}
```

Menghubungkan Route ke Controller:

Sekarang Anda dapat mendefinisikan *route* yang menunjuk ke metode *controller* ini:

```
// file: routes/web.php

use App\Http\Controllers\UserController;

Route::get(
    '/user/{id}',
    [UserController::class, 'show']
);
```

Ketika *request* yang cocok dengan URI *route* ini diterima, metode `show()` di kelas `UserController` akan dieksekusi. Parameter *route* juga akan diteruskan ke metode tersebut.

5.2. Middleware

Middleware dapat diterapkan ke *route controller* di file *route* Anda, atau Anda dapat menetapkannya di dalam konstruktor *controller* Anda. Menggunakan metode `middleware()` di dalam konstruktor *controller* Anda, Anda dapat dengan mudah menetapkan *middleware* ke *action controller*.

Contoh:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class AdminController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('admin');
    }

    // ...
}
```

Anda juga dapat membatasi *middleware* hanya untuk metode tertentu di dalam *controller*.

```
$this->middleware('auth')->only('store', 'update', 'destroy');

$this->middleware('subscribed')->except('show');
```

5.3. Resource Controller

Jika Anda membangun aplikasi CRUD (*Create, Read, Update, Delete*), Anda mungkin akan menemukan diri Anda membuat *controller* yang memiliki metode untuk setiap operasi CRUD. Laravel menyediakan cara mudah untuk membuat *controller* semacam ini dengan satu perintah Artisan.

Membuat Resource Controller:

```
php artisan make:controller PhotoController --resource
```

Perintah ini akan membuat *controller* di `app/Http/Controllers/PhotoController.php` yang berisi metode untuk setiap operasi sumber daya yang tersedia:

- `index()`: Menampilkan daftar sumber daya.
- `create()`: Menampilkan formulir untuk membuat sumber daya baru.
- `store(Request $request)`: Menyimpan sumber daya baru ke *database*.
- `show($id)`: Menampilkan sumber daya tertentu.
- `edit($id)`: Menampilkan formulir untuk mengedit sumber daya tertentu.
- `update(Request` $request$`,id)`: Memperbarui sumber daya tertentu di *database*.
- `destroy($id)`: Menghapus sumber daya tertentu dari *database*.

Mendaftarkan Resource Route:

Untuk mendaftarkan *resource route*, Anda dapat menggunakan metode `resource()` pada fasad `Route`:

```
// file: routes/web.php

use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Panggilan metode tunggal ini membuat beberapa *route* untuk menangani berbagai tindakan pada sumber daya. Anda dapat melihat daftar lengkap *route* yang dibuat dengan menjalankan perintah `php artisan route:list`.

5.4. Dependency Injection

Laravel menggunakan *service container* untuk mengelola semua dependensi kelas. *Dependency injection* adalah cara untuk "menyuntikkan" dependensi ke dalam kelas melalui konstruktor atau metode "setter".

Constructor Injection

Anda dapat mengetik-hint dependensi yang Anda butuhkan di konstruktor *controller*. Laravel akan secara otomatis me-*resolve* dan menyuntikkan dependensi tersebut ke dalam instance *controller*.

Contoh:

```php
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $`this->users = `$users;
    }

    // ...
}
```

Method Injection

Selain *constructor injection*, Anda juga dapat mengetik-hint dependensi pada metode *controller* Anda. Dependensi akan di-*resolve* dan disuntikkan hanya untuk panggilan metode tersebut.

Contoh:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->name;

        //
    }
}
```

5.5. Request

Untuk mendapatkan informasi tentang *request* HTTP saat ini, Anda dapat menggunakan fasad `Request` atau menyuntikkan instance `Illuminate\Http\Request` ke metode *controller* Anda.

Mengakses Request:

```
// Menggunakan fasad Request
use Illuminate\Support\Facades\Request;

$name = Request::input('name');

// Menggunakan dependency injection
public function store(Request $request)
{
    $name = $request->input('name');
}
```

Objek `Request` menyediakan berbagai metode untuk memeriksa *request* HTTP, seperti:

- `$request->path()` : Mendapatkan path *request*.
- `$request->url()` : Mendapatkan URL lengkap.
- `$request->method()` : Mendapatkan metode *request* (GET, POST, dll.).
- `$request->isMethod('post')` : Memeriksa apakah metode *request* adalah POST.
- `$request->header('Content-Type')` : Mendapatkan header *request*.
- `$request->bearerToken()` : Mendapatkan token otentikasi.
- `$request->ip()` : Mendapatkan alamat IP klien.

5.6. Response

Semua *route* dan *controller* harus mengembalikan *response* untuk dikirim kembali ke *browser* pengguna. Laravel menyediakan beberapa cara berbeda untuk mengembalikan *response*.

String & Array

Response paling dasar adalah mengembalikan string atau *array* dari *route* atau *controller*. Laravel akan secara otomatis mengubahnya menjadi *response* HTTP yang sesuai.

```
Route::get(
    '/',
    function () {
        return 'Hello World';
    }
);

Route::get(
    '/users',
    function () {
        return [
            ['name' => 'John Doe'],
            ['name' => 'Jane Doe'],
        ];
    }
);
```

Response Object

Biasanya, Anda tidak akan hanya mengembalikan string atau *array* sederhana. Sebaliknya, Anda akan mengembalikan instance `Illuminate\Http\Response` penuh atau *view*.

```
use Illuminate\Http\Response;

Route::get(
    '/home',
    function () {
        return response('Hello World', 200)
            ->header('Content-Type', 'text/plain');
    }
);
```

Redirect

Redirect response adalah instance dari kelas `Illuminate\Http\RedirectResponse`, dan berisi header yang diperlukan untuk mengarahkan pengguna ke URL lain. Ada beberapa cara untuk menghasilkan instance `RedirectResponse`.

```
// Redirect ke URI tertentu
return redirect('home/dashboard');

// Redirect ke named route
return redirect()->route('profile');

// Redirect ke named route dengan parameter
return redirect()->route('profile', ['id' => 1]);

// Redirect kembali ke halaman sebelumnya
return back()->withInput();
```

---

6. Laravel View

Dalam arsitektur MVC, *View* bertanggung jawab untuk menyajikan data kepada pengguna. Laravel menyediakan sistem *templating* yang kuat dan intuitif yang disebut Blade. Blade memungkinkan Anda menulis *markup view* yang bersih dan mudah dibaca, sambil tetap memberikan kekuatan penuh dari PHP.

6.1. Blade Templates

Semua file *view* Blade memiliki ekstensi `.blade.php` dan biasanya disimpan di direktori `resources/views`. Anda dapat menggunakan fungsi `view()` global untuk mengembalikan *view* dari *route* atau *controller* Anda.

Contoh:

```
// file: routes/web.php

use Illuminate\Support\Facades\Route;

Route::get(
    '/greeting',
    function () {
        return view('welcome', ['name' => 'John Doe']);
    }
);
```

Dalam contoh di atas, kita mengembalikan *view* `welcome.blade.php` dan meneruskan variabel `name` ke *view* tersebut.

Menampilkan Data

Anda dapat menampilkan data yang diteruskan ke *view* Blade dengan menyertakan variabel dalam kurung kurawal ganda `{{ }}`. Ini secara otomatis akan membersihkan data untuk mencegah serangan XSS.

```
<!-- file: resources/views/welcome.blade.php -->

<h1>Hello, {{ $name }}!</h1>
```

Struktur Kontrol Blade

Blade menyediakan sejumlah *directive* yang nyaman untuk struktur kontrol umum seperti *conditional statement* dan *loop*.

If Statements:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif

@unless (Auth::check())
    You are not signed in.
@endunless
```

Loops:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

Layout Inheritance

Salah satu fitur paling kuat dari Blade adalah *layout inheritance*. Ini memungkinkan Anda untuk mendefinisikan *layout* master dan kemudian memiliki halaman anak yang mewarisi *layout* tersebut. Ini sangat berguna untuk menjaga konsistensi desain di seluruh aplikasi Anda.

Mendefinisikan Layout Master:

Buat file `resources/views/layouts/app.blade.php`:

```blade
<!-- file: resources/views/layouts/app.blade.php -->

<!DOCTYPE html>
<html lang="en">
<head>
    <title>App Name - @yield('title')</title>
</head>
<body>
    @section('sidebar')
        This is the master sidebar.
    @show

    <div class="container">
        @yield('content')
    </div>
</body>
</html>
```

- `@yield('title')`: Mendefinisikan bagian yang dapat diisi oleh halaman anak.
- `@section('sidebar') ... @show`: Mendefinisikan bagian dengan konten *default* yang dapat ditimpa atau ditambahkan oleh halaman anak.

Memperluas Layout Master:

Sekarang, buat halaman anak yang akan menggunakan *layout* ini, misalnya `resources/views/child.blade.php`:

```blade
<!-- file: resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('content')
    <p>This is my body content.</p>
@endsection

@section('sidebar')
    @parent
    <p>This is appended to the master sidebar.</p>
@endsection
```

- `@extends('layouts.app')`: Menentukan bahwa *view* ini mewarisi dari `layouts.app`.
- `@section('title', 'Page Title')`: Mengisi bagian `title`.
- `@section('content') ... @endsection`: Mengisi bagian `content`.
- `@parent`: Digunakan di dalam `@section` untuk menyertakan konten dari bagian *parent* sebelum menambahkan konten baru.

Komponen & Slot

Laravel juga mendukung komponen Blade dan slot, yang menyediakan pendekatan yang berbeda untuk *layout* dan reusable elemen. Ini memungkinkan Anda untuk membuat komponen UI kecil yang dapat digunakan kembali di seluruh aplikasi Anda.

Membuat Komponen:

```
php artisan make:component Alert
```

Ini akan membuat dua file: `app/View/Components/Alert.php` dan `resources/views/components/alert.blade.php`.

Menggunakan Komponen:

```
<!-- file: resources/views/welcome.blade.php -->

<x-alert type="error" :message="$message"/>

<x-alert type="success">
    <strong>Success!</strong> Your operation was successful.
</x-alert>
```

Definisi Komponen (resources/views/components/alert.blade.php):

```
<!-- file: resources/views/components/alert.blade.php -->

<div class="alert alert-{{ $type }}">
    {{ $message ?? $slot }}
</div>
```

Blade menyediakan cara yang sangat fleksibel dan kuat untuk membangun *view* yang dinamis dan mudah dipelihara di aplikasi Laravel Anda.

---

7. Laravel Database

Laravel menyediakan berbagai *tool* dan fitur untuk berinteraksi dengan *database* Anda, mulai dari konfigurasi koneksi hingga manajemen skema *database* dan manipulasi data. Laravel mendukung beberapa sistem *database* seperti MySQL, PostgreSQL, SQLite, dan SQL Server.

7.1. Setting Database

Konfigurasi *database* utama untuk aplikasi Laravel Anda terletak di file `.env` di *root* proyek Anda. File ini berisi variabel lingkungan yang digunakan untuk mengkonfigurasi koneksi *database* Anda.

Contoh Konfigurasi di `.env` :

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

Pastikan Anda mengganti nilai-nilai ini sesuai dengan pengaturan *database* lokal atau *server* Anda. Setelah mengubah file `.env`, Anda mungkin perlu membersihkan *cache* konfigurasi Laravel dengan perintah:

```
php artisan config:clear
```

7.2. Database Migration

Migration adalah fitur Laravel yang memungkinkan Anda mengelola skema *database* aplikasi Anda dengan cara yang terstruktur dan terorganisir. *Migration* seperti kontrol versi untuk *database* Anda, memungkinkan Anda dan tim Anda untuk dengan mudah memodifikasi dan berbagi skema *database* aplikasi.

Membuat Migration:

Anda dapat membuat file *migration* baru menggunakan perintah Artisan `make:migration`:

```
php artisan make:migration create_users_table
```

Perintah ini akan membuat file *migration* baru di direktori `database/migrations`. File *migration* berisi dua metode: `up()` dan `down()`.

- Metode `up()` digunakan untuk menambahkan tabel, kolom, atau indeks baru ke *database* Anda.

- Metode `down()` digunakan untuk membalikkan operasi yang dilakukan oleh metode `up()`, yang berguna saat melakukan *rollback migration*.

Contoh Migration:

```
// file: database/migrations/YYYY_MM_DD_HHMMSS_create_users_table.php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create(
            'users',
            function (Blueprint $table) {
                $table->id();
                $table->string('name');
                $table->string('email')->unique();
                $table->timestamp('email_verified_at')->nullable();
                $table->string('password');
                $table->rememberToken();
                $table->timestamps();
            }
        );
    }

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

Menjalankan Migration:

Setelah Anda membuat file *migration*, Anda dapat menjalankannya menggunakan perintah Artisan `migrate`:

```
php artisan migrate
```

Perintah ini akan menjalankan semua *migration* yang belum dijalankan dan membuat tabel di *database* Anda.

Rollback Migration:

Jika Anda perlu mengembalikan *migration* terakhir, Anda dapat menggunakan perintah `migrate:rollback`:

```
php artisan migrate:rollback
```

7.3. Seeder

Seeder digunakan untuk mengisi *database* Anda dengan data *dummy* atau data awal yang diperlukan untuk pengujian atau pengembangan. Ini sangat berguna untuk mengisi *database* dengan data yang konsisten setiap kali Anda membangun ulang *database* Anda.

Membuat Seeder:

Anda dapat membuat file *seeder* baru menggunakan perintah Artisan `make:seeder`:

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

Ini akan membuat file baru di `database/seeders/UserSeeder.php`:

```
// file: database/seeders/UserSeeder.php

namespace Database\Seeders;

use App\Models\User;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Hash;

class UserSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        User::create([
            'name' => 'Admin User',
            'email' => 'admin@example.com',
            'password' => Hash::make('password'),
        ]);

        User::factory()->count(10)->create();
    }
}
```

Menjalankan Seeder:

Untuk menjalankan *seeder*, Anda dapat menggunakan perintah Artisan `db:seed`:

```
php artisan db:seed
```

Jika Anda ingin menjalankan *seeder* tertentu, Anda dapat menggunakan opsi `--class`:

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

Anda juga dapat menjalankan *migration* dan *seeder* secara bersamaan dengan perintah `migrate:fresh --seed` (ini akan menghapus semua tabel dan menjalankan *migration* dari awal, lalu menjalankan *seeder*).

7.4. Query Builder

Laravel Query Builder menyediakan antarmuka yang nyaman dan ekspresif untuk membuat dan menjalankan *query database*. Ini dapat digunakan untuk melakukan sebagian besar operasi *database* dalam aplikasi Anda dan bekerja dengan semua sistem *database* yang didukung Laravel.

Mengambil Data:

```
// Mengambil semua baris dari tabel
$users = DB::table('users')->get();

// Mengambil satu baris berdasarkan kondisi
$user = DB::table('users')->where('name', 'John')->first();

// Mengambil nilai kolom tertentu
$email = DB::table('users')->where('name', 'John')->value('email');

// Mengambil daftar nilai kolom
$titles = DB::table('posts')->pluck('title');

// Menggunakan SELECT tertentu
$users = DB::table('users')->select('name', 'email')->get();
```

Menyisipkan Data:

```
DB::table('users')->insert([
    'name' => 'Jane Doe',
    'email' => 'jane@example.com',
    'password' => Hash::make('password'),
]);
```

Memperbarui Data:

```
DB::table('users')
    ->where('id', 1)
    ->update(['name' => 'Updated Name']);
```

Menghapus Data:

```
DB::table('users')->where('id', 1)->delete();

DB::table('users')->delete(); // Menghapus semua baris
```

Join Table:

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

Query Builder sangat fleksibel dan memungkinkan Anda untuk membangun *query* yang kompleks dengan cara yang mudah dibaca dan dipelihara.

---

8. Laravel Eloquent

Eloquent ORM adalah *Object-Relational Mapper (ORM)* yang disertakan dengan Laravel. Eloquent menyediakan cara yang indah dan ekspresif untuk berinteraksi dengan *database* Anda. Setiap tabel *database* memiliki "Model" yang sesuai, yang digunakan untuk berinteraksi dengan tabel tersebut. Model memungkinkan Anda untuk melakukan *query* pada tabel Anda serta menyisipkan, memperbarui, dan menghapus *record* baru dari tabel.

8.1. Model

Secara *default*, model Eloquent berada di direktori `app/Models`. Anda dapat membuat model baru menggunakan perintah Artisan `make:model`:

```
php artisan make:model Post
```

Ini akan membuat file `app/Models/Post.php`:

```php
<?php

namespace App\Models;

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

class Post extends Model
{
    use HasFactory;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'title',
        'content',
    ];
}
```

Secara *default*, model Eloquent akan mengasumsikan bahwa tabel *database* yang sesuai dengan model adalah bentuk jamak (plural) dari nama model. Jadi, model `Post` akan berinteraksi dengan tabel `posts`.

Mengambil Data

```php
// Mengambil semua record
$posts = Post::all();

// Mengambil record berdasarkan primary key
$post = Post::find(1);

// Mengambil record pertama yang cocok dengan kondisi
$post = Post::where("title", "My First Post")->first();

// Mengambil semua record yang cocok dengan kondisi
$posts = Post::where("published", true)->get();

// Mengambil record dengan batasan
$posts = Post::where("views", ">", 100)->orderBy("created_at", "desc")-
>take(10)->get();
```

Menyisipkan Data

```
 $post = new Post;
$post->title = "New Post Title";
$post->content = "This is the content of the new post.";
$post->save();

// Atau menggunakan mass assignment (pastikan $fillable diatur di model)
$post = Post::create([
    "title" => "Another New Post",
    "content" => "Content for another new post.",
]);
```

Memperbarui Data

```
 $post = Post::find(1);
$post->title = "Updated Post Title";
$post->save();

// Atau menggunakan mass update
Post::where("published", false)->update(["published" => true]);
```

Menghapus Data

```
 $post = Post::find(1);
$post->delete();

// Atau menghapus berdasarkan primary key
Post::destroy(1);

// Atau menghapus berdasarkan kondisi
Post::where("published", false)->delete();
```

8.2. Relationship

Salah satu fitur paling kuat dari Eloquent adalah kemampuannya untuk mengelola hubungan antar tabel *database*. Eloquent mendukung berbagai jenis hubungan:

One To One

Hubungan *one-to-one* adalah hubungan yang sangat dasar. Misalnya, model `User` mungkin memiliki satu `Phone`.

```
// User Model
class User extends Model
{
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}

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

Penggunaan:

```
$phone = User::find(1)->phone;
$user = Phone::find(1)->user;
```

One To Many

Hubungan *one-to-many* digunakan untuk mendefinisikan hubungan di mana satu model adalah *parent* dari satu atau lebih model anak. Misalnya, sebuah `Post` mungkin memiliki banyak `Comment`.

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

// Comment Model
class Comment extends Model
{
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Penggunaan:

```
$comments = Post::find(1)->comments;
$post = Comment::find(1)->post;
```

Many To Many

Hubungan *many-to-many* sedikit lebih kompleks. Misalnya, seorang `User` mungkin memiliki banyak `Role`, dan sebuah `Role` mungkin dimiliki oleh banyak `User`. Hubungan ini memerlukan tabel *pivot* perantara.

```
 // User Model
class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}

// Role Model
class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Penggunaan:

```
 $roles = User::find(1)->roles;
$users = Role::find(1)->users;

// Menambahkan role ke user
$`user->roles()->attach(`$roleId);

// Melepaskan role dari user
$`user->roles()->detach(`$roleId);

// Menyinkronkan role (menambahkan/menghapus sesuai array yang diberikan)
$user->roles()->sync([1, 2, 3]);
```

Has Many Through

Hubungan "has many through" menyediakan cara mudah untuk mengakses hubungan yang lebih jauh melalui model perantara. Misalnya, model `Country` mungkin memiliki banyak `Post` melalui model `User`.

```
 // Country Model
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(Post::class, User::class);
    }
}

// User Model (dengan country_id)
class User extends Model
{
    public function country()
    {
        return $this->belongsTo(Country::class);
    }
}

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

Penggunaan:

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

Eloquent ORM dan fitur *relationship*-nya adalah salah satu alasan utama mengapa Laravel begitu disukai oleh *developer*. Ini menyederhanakan interaksi *database* dan memungkinkan Anda untuk fokus pada logika bisnis aplikasi Anda.

---

9. Model, View, Controller Workflow

Arsitektur Model-View-Controller (MVC) adalah pola desain perangkat lunak yang memisahkan aplikasi menjadi tiga komponen utama yang saling terhubung. Laravel, sebagai *framework* MVC, mengimplementasikan pola ini dengan sangat baik, memungkinkan *developer* untuk membangun aplikasi yang terstruktur, mudah dipelihara, dan dapat diskalakan. Memahami bagaimana ketiga komponen ini berinteraksi adalah kunci untuk mengembangkan aplikasi Laravel yang efektif.

9.1. Peran Masing-Masing Komponen

- **Model:**
    - Merepresentasikan data dan logika bisnis yang terkait dengan data tersebut.
    - Berinteraksi langsung dengan *database* (melalui Eloquent ORM di Laravel).
    - Bertanggung jawab untuk mengambil, menyimpan, memperbarui, dan menghapus data.
    - Dapat berisi validasi data dan aturan bisnis.
- **View:**
    - Bertanggung jawab untuk presentasi data kepada pengguna.
    - Tidak berisi logika bisnis atau interaksi *database*.
    - Di Laravel, *view* biasanya dibuat menggunakan Blade *templating engine* (`.blade.php`).
    - Menerima data dari *controller* dan menampilkannya dalam format yang sesuai (HTML, JSON, dll.).
- **Controller:**
    - Bertindak sebagai perantara antara Model dan View.
    - Menerima *request* dari pengguna (melalui *route*).
    - Memproses *request* tersebut, seringkali dengan berinteraksi dengan Model untuk mengambil atau memanipulasi data.
    - Memilih *view* yang sesuai untuk menampilkan *response* dan meneruskan data yang diperlukan ke *view* tersebut.
    - Dapat berisi logika untuk menangani *input* pengguna, validasi, dan *redirect*.

9.2. Alur Kerja MVC dalam Laravel

Mari kita jelajahi alur kerja tipikal *request* dalam aplikasi Laravel yang mengikuti pola MVC:

1. **Request Masuk (HTTP Request):**

- Pengguna membuka *browser* dan mengetikkan URL (misalnya, `http://your-app.com/products/1`).
- *Request* HTTP ini dikirim ke *web server* (Apache/Nginx) yang kemudian meneruskannya ke aplikasi Laravel.

2. **Routing:**
   - Laravel menerima *request* dan mencocokkannya dengan *route* yang telah didefinisikan di file `routes/web.php` (atau `api.php`, dll.).
   - *Route* akan menentukan *controller* dan metode mana yang harus menangani *request* ini.

   Contoh Route: ```php // file: routes/web.php use App\Http\Controllers\ProductController;

   Route::get('/products/{id}', [ProductController::class, 'show']); ```

3. **Controller Action:**
   - Setelah *route* cocok, metode `show` di `ProductController` akan dipanggil.
   - *Controller* akan menerima parameter dari *route* (dalam hal ini, `id` produk).

   Contoh Controller: ```php // file: app/Http/Controllers/ProductController.php <?php

   namespace App\Http\Controllers;

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

   class ProductController extends Controller { public function show($id) { // 1. Controller berinteraksi dengan Model $ `product = Product::findOrFail(` $id);

```
    // 2. Controller meneruskan data ke View
    return view(
        'products.show',
        compact('product')
    );
}
```

   } ```

4. **Interaksi dengan Model:**

- Di dalam *controller*, logika bisnis dimulai. *Controller* akan meminta data dari Model.
- Dalam contoh di atas, `Product::findOrFail($id)` akan mencari produk dengan ID yang diberikan dari *database* menggunakan Model `Product` (Eloquent ORM).
- Model akan berinteraksi dengan *database* dan mengembalikan objek `Product` yang berisi data produk.

5. **View Rendering:**
    - Setelah *controller* mendapatkan data yang diperlukan dari Model, ia akan memilih *view* yang sesuai untuk menampilkan data tersebut.
    - *Controller* meneruskan data (`$product`) ke *view* `products.show`.

    Contoh View: ```blade

{{ $product->name }}

{{ $product->description }}

Price: $ {{ $product->price }}

```

6. **Response Kembali ke Pengguna:**
    - Blade *templating engine* akan memproses *view* `products.show.blade.php`, menggabungkan data `$product` ke dalam *markup* HTML.
    - HTML yang dihasilkan dikirim kembali melalui *controller* dan *route* sebagai *response* HTTP ke *browser* pengguna.
    - Pengguna melihat halaman produk yang dirender di *browser* mereka.
```

9.3. Keuntungan Pola MVC

- **Pemisahan Tanggung Jawab (Separation of Concerns):** Setiap komponen memiliki tanggung jawab yang jelas, membuat kode lebih terorganisir dan mudah dipahami.
- **Kemudahan Pemeliharaan:** Perubahan pada satu bagian aplikasi (misalnya, *database* atau *UI*) cenderung tidak memengaruhi bagian lain secara signifikan.
- **Reusability:** Model dan *controller* dapat digunakan kembali di berbagai bagian aplikasi atau bahkan di proyek lain.
- **Pengujian:** Memisahkan logika memungkinkan pengujian unit yang lebih mudah untuk setiap komponen.
- **Kolaborasi:** Beberapa *developer* dapat bekerja pada bagian yang berbeda dari aplikasi secara bersamaan tanpa banyak konflik.

Memahami alur kerja MVC ini sangat penting untuk membangun aplikasi Laravel yang terstruktur dengan baik dan efisien.

---

10. SQLite

SQLite adalah sistem manajemen *database* relasional yang ringan, *self-contained*, dan tidak memerlukan *server* terpisah. Ini sangat populer untuk aplikasi *desktop*, aplikasi seluler, dan juga sangat berguna untuk pengembangan aplikasi web lokal, *testing*, dan aplikasi skala kecil di Laravel. Laravel memiliki dukungan *built-in* untuk SQLite, membuatnya mudah untuk digunakan.

10.1. Mengapa Menggunakan SQLite di Laravel?

- **Kemudahan Setup:** Tidak perlu menginstal dan mengkonfigurasi *server database* terpisah seperti MySQL atau PostgreSQL. Cukup buat file *database* tunggal.
- **Portabilitas:** File *database* SQLite dapat dengan mudah disalin dan dipindahkan antar lingkungan.
- **Pengujian:** Sangat ideal untuk *testing* otomatis karena *database* dapat dengan mudah dibuat ulang untuk setiap *test*.

- **Pengembangan Lokal:** Cocok untuk pengembangan lokal di mana *developer* tidak ingin mengelola *server database* yang kompleks.
- **Aplikasi Skala Kecil:** Cukup untuk aplikasi dengan volume data dan *traffic* yang rendah.

10.2. Konfigurasi SQLite di Laravel

Untuk menggunakan SQLite di Laravel, Anda perlu melakukan beberapa konfigurasi sederhana.

a. Membuat File Database SQLite

Secara *default*, Laravel mencari file *database* SQLite di direktori `database`. Anda bisa membuat file kosong bernama `database.sqlite` di direktori `database` proyek Laravel Anda.

```
# Dari root proyek Laravel Anda
touch database/database.sqlite
```

b. Mengkonfigurasi `.env`

Selanjutnya, Anda perlu memperbarui file `.env` untuk menunjuk ke koneksi SQLite. Ubah baris `DB_CONNECTION` dan hapus atau komentari baris lain yang terkait dengan koneksi *database* lain (seperti MySQL).

```
DB_CONNECTION=sqlite
# DB_HOST=127.0.0.1
# DB_PORT=3306
# DB_DATABASE=laravel_db
# DB_USERNAME=root
# DB_PASSWORD=
```

Dengan `DB_CONNECTION=sqlite`, Laravel akan secara otomatis mencari file `database.sqlite` di direktori `database/`.

c. Menjalankan Migration

Setelah konfigurasi selesai, Anda dapat menjalankan *migration* Anda untuk membuat tabel di *database* SQLite Anda, sama seperti yang Anda lakukan dengan *database* lainnya:

```
 bash
php artisan migrate
```

Ini akan membuat semua tabel yang didefinisikan dalam file *migration* Anda di dalam file `database/database.sqlite`.

10.3. Menggunakan SQLite untuk Testing

SQLite sangat sering digunakan untuk *testing* di Laravel karena kemudahannya untuk di-*reset* dan diisi dengan data *dummy*.

Secara *default*, file konfigurasi `config/database.php` Laravel memiliki koneksi `sqlite` yang dikonfigurasi untuk menggunakan *database* dalam memori (`:memory:`) saat menjalankan *test*.

```
// file: config/database.php

        'sqlite' => [
            'driver' => 'sqlite',
            'url' => env('DB_URL'),
            'database' => env('DB_DATABASE', database_path('database.sqlite')),
            'prefix' => '',
            'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),
        ],
```

Ketika Anda menjalankan *test* Laravel, Anda dapat menggunakan *trait* `RefreshDatabase` di *test case* Anda. *Trait* ini akan secara otomatis me-*reset database* untuk setiap *test*, memastikan bahwa setiap *test* berjalan di lingkungan *database* yang bersih.

```php
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class UserTest extends TestCase
{
    use RefreshDatabase; // Ini akan me-reset database untuk setiap test

    /**
     * A basic feature test example.
     */
    public function test_example(): void
    {
        $`response = `$this->get(
            /
        );

        $response->assertStatus(200);
    }
}
```

Dengan `RefreshDatabase`, Anda tidak perlu khawatir tentang *state database* antar *test*, karena setiap *test* akan dimulai dengan *database* yang baru dan kosong.

10.4. Keterbatasan SQLite

Meskipun SQLite sangat nyaman, penting untuk diingat bahwa ia memiliki beberapa keterbatasan dibandingkan dengan *database server* penuh seperti MySQL atau PostgreSQL:

- **Concurrency:** SQLite tidak dirancang untuk *concurrency* tinggi. Ini mengunci seluruh file *database* saat menulis, yang dapat menyebabkan masalah performa di lingkungan multi-pengguna atau aplikasi web dengan *traffic* tinggi.
- **Skalabilitas:** Tidak cocok untuk aplikasi yang membutuhkan skalabilitas besar atau menangani volume data yang sangat besar.
- **Fitur Lanjutan:** Beberapa fitur *database* tingkat lanjut yang ditemukan di *database server* mungkin tidak tersedia atau tidak diimplementasikan dengan cara yang sama di SQLite.

Secara keseluruhan, SQLite adalah pilihan yang sangat baik untuk pengembangan lokal, *testing*, dan aplikasi Laravel skala kecil yang tidak memerlukan *database server* yang kompleks.

---

11. Make Simple Project in Laravel

Setelah memahami dasar-dasar Laravel, mari kita coba membuat proyek sederhana untuk mengaplikasikan konsep-konsep yang telah dipelajari. Kita akan membuat aplikasi daftar tugas (To-Do List) sederhana.

11.1. Persiapan Proyek Baru

Jika Anda belum memiliki proyek Laravel, buatlah proyek baru:

```
composer create-project laravel/laravel todo-app
cd todo-app
php artisan serve
```

Pastikan aplikasi berjalan dengan baik di `http://127.0.0.1:8000`.

11.2. Konfigurasi Database

Untuk proyek sederhana ini, kita akan menggunakan SQLite untuk kemudahan. Buat file `database.sqlite` di direktori `database/`:

```
touch database/database.sqlite
```

Kemudian, ubah konfigurasi `.env` Anda:

```
DB_CONNECTION=sqlite
# DB_HOST=127.0.0.1
# DB_PORT=3306
# DB_DATABASE=laravel_db
# DB_USERNAME=root
# DB_PASSWORD=
```

11.3. Membuat Model dan Migration

Kita akan membuat model `Task` dan *migration* untuk tabel `tasks`.

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

Perintah `-m` akan membuat file *migration* bersamaan dengan model. Buka file *migration* yang baru dibuat di

`database/migrations/YYYY_MM_DD_HHMMSS_create_tasks_table.php` dan tambahkan kolom `description` dan `completed`:

```
// file: database/migrations/YYYY_MM_DD_HHMMSS_create_tasks_table.php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create(
            'tasks',
            function (Blueprint $table) {
                $table->id();
                $table->string('description');
                $table->boolean('completed')->default(false);
                $table->timestamps();
            }
        );
    }

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

Jalankan *migration* untuk membuat tabel `tasks` di *database* Anda:

```
php artisan migrate
```

11.4. Membuat Controller

Kita akan membuat `TaskController` untuk menangani logika aplikasi.

```
php artisan make:controller TaskController --resource
```

Ini akan membuat *resource controller* dengan metode `index`, `create`, `store`, `show`, `edit`, `update`, dan `destroy`. Untuk proyek sederhana ini, kita akan fokus pada `index` (menampilkan daftar tugas) dan `store` (menambahkan tugas baru).

Edit `app/Http/Controllers/TaskController.php`:

```
// file: app/Http/Controllers/TaskController.php

<?php

namespace App\Http\Controllers;

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

class TaskController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        $tasks = Task::all();
        return view(
            'tasks.index',
            compact('tasks')
        );
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $request->validate([
            'description' => 'required|max:255',
        ]);

        Task::create([
            'description' => $request->description,
        ]);

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

    // Metode lain seperti create, show, edit, update, destroy dapat
ditambahkan nanti
}
```

11.5. Mendefinisikan Route

Dafrarkan *resource route* untuk `TaskController` di `routes/web.php`:

```
// file: routes/web.php

use App\Http\Controllers\TaskController;
use Illuminate\Support\Facades\Route;

Route::get(
    '/',
    function () {
        return redirect()->route('tasks.index');
    }
);

Route::resource('tasks', TaskController::class);
```

Dengan ini, *route* `/` akan me-*redirect* ke `/tasks`.

11.6. Membuat View

Buat direktori `resources/views/tasks` dan di dalamnya buat file `index.blade.php`.

```blade
<!-- file: resources/views/tasks/index.blade.php -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do List</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        .container { max-width: 600px; margin: auto; background: #f9f9f9;
padding: 20px; border-radius: 8px; box-shadow: 0 0 10px rgba(0,0,0,0.1); }
        h1 { text-align: center; color: #333; }
        form { display: flex; margin-bottom: 20px; }
        form input[type="text"] { flex-grow: 1; padding: 10px; border: 1px
solid #ddd; border-radius: 4px 0 0 4px; }
        form button { padding: 10px 15px; background: #007bff; color: white;
border: none; border-radius: 0 4px 4px 0; cursor: pointer; }
        form button:hover { background: #0056b3; }
        ul { list-style: none; padding: 0; }
        li { background: white; padding: 10px; margin-bottom: 8px; border-
radius: 4px; border: 1px solid #eee; display: flex; justify-content: space-
between; align-items: center; }
        li.completed { text-decoration: line-through; color: #888; }
        .error { color: red; font-size: 0.9em; margin-top: 5px; }
    </style>
</head>
<body>
    <div class="container">
        <h1>My To-Do List</h1>

        <form action="{{ route('tasks.store') }}" method="POST">
            @csrf
            <input type="text" name="description" placeholder="Add a new
task..." value="{{ old('description') }}">
            <button type="submit">Add Task</button>
        </form>

        @error('description')
            <div class="error">{{ $message }}</div>
        @enderror

        <ul>
            @forelse ($`tasks as `$task)
                <li class="{{ $task->completed ? 'completed' : '' }}">
                    <span>{{ $task->description }}</span>
                    <!-- Tambahkan tombol untuk mark as complete/delete di sini
nanti -->
                </li>
            @empty
                <li>No tasks yet. Add one!</li>
            @endforelse
        </ul>
    </div>
</body>
</html>
```

11.7. Menguji Aplikasi

Sekarang, buka `http://127.0.0.1:8000` di *browser* Anda. Anda akan melihat halaman daftar tugas. Anda dapat menambahkan tugas baru, dan tugas tersebut akan muncul di daftar. Ini adalah contoh sederhana bagaimana Model, View, dan Controller bekerja sama dalam aplikasi Laravel.

---

12. CRUD Application

Melanjutkan dari proyek sederhana sebelumnya, kita akan mengembangkan aplikasi To-Do List menjadi aplikasi CRUD (Create, Read, Update, Delete) yang lengkap. Ini akan melibatkan penambahan fungsionalitas untuk mengedit dan menghapus tugas.

12.1. Memperbarui Model Task

Pastikan model `Task` Anda memiliki properti `$fillable` yang memungkinkan *mass assignment* untuk kolom `description` dan `completed`.

```
// file: app/Models/Task.php

<?php

namespace App\Models;

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

class Task extends Model
{
    use HasFactory;

    protected $fillable = [
        'description',
        'completed',
    ];
}
```

12.2. Memperbarui TaskController

Kita akan menambahkan logika untuk metode `update` dan `destroy` di `TaskController`.

```php
// file: app/Http/Controllers/TaskController.php

<?php

namespace App\Http\Controllers;

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

class TaskController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        $tasks = Task::all();
        return view(
            'tasks.index',
            compact('tasks')
        );
    }

    /**
     * Store a newly created resource in storage.
     */
    public function store(Request $request)
    {
        $request->validate([
            'description' => 'required|max:255',
        ]);

        Task::create([
            'description' => $request->description,
        ]);

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

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(Task $task)
    {
        return view('tasks.edit', compact('task'));
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $`request, Task `$task)
    {
        $request->validate([
            'description' => 'required|max:255',
        ]);

        $task->update([
            'description' => $request->description,
            'completed' => $request->has('completed'), // Checkbox value
        ]);
```

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

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(Task $task)
    {
        $task->delete();

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

12.3. Memperbarui View `index.blade.php`

Kita akan menambahkan tombol Edit dan Delete untuk setiap tugas di `resources/views/tasks/index.blade.php`.

```
<!-- file: resources/views/tasks/index.blade.php -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do List</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        .container { max-width: 600px; margin: auto; background: #f9f9f9;
padding: 20px; border-radius: 8px; box-shadow: 0 0 10px rgba(0,0,0,0.1); }
        h1 { text-align: center; color: #333; }
        form { display: flex; margin-bottom: 20px; }
        form input[type="text"] { flex-grow: 1; padding: 10px; border: 1px
solid #ddd; border-radius: 4px 0 0 4px; }
        form button { padding: 10px 15px; background: #007bff; color: white;
border: none; border-radius: 0 4px 4px 0; cursor: pointer; }
        form button:hover { background: #0056b3; }
        ul { list-style: none; padding: 0; }
        li { background: white; padding: 10px; margin-bottom: 8px; border-
radius: 4px; border: 1px solid #eee; display: flex; justify-content: space-
between; align-items: center; }
        li.completed { text-decoration: line-through; color: #888; }
        .actions button { margin-left: 5px; padding: 5px 10px; border-radius:
4px; cursor: pointer; }
        .actions .edit-btn { background: #ffc107; color: white; border: none; }
        .actions .edit-btn:hover { background: #e0a800; }
        .actions .delete-btn { background: #dc3545; color: white; border: none;
}
        .actions .delete-btn:hover { background: #c82333; }
        .error { color: red; font-size: 0.9em; margin-top: 5px; }
    </style>
</head>
<body>
    <div class="container">
        <h1>My To-Do List</h1>

        <form action="{{ route('tasks.store') }}" method="POST">
            @csrf
            <input type="text" name="description" placeholder="Add a new
task..." value="{{ old('description') }}">
            <button type="submit">Add Task</button>
        </form>

        @error('description')
            <div class="error">{{ $message }}</div>
        @enderror

        <ul>
            @forelse ($`tasks as `$task)
                <li class="{{ $task->completed ? 'completed' : '' }}">
                    <span>{{ $task->description }}</span>
                    <div class="actions">
                        <a href="{{ route('tasks.edit', $task->id) }}"
class="edit-btn">Edit</a>
                        <form action="{{ route('tasks.destroy', $task->id) }}"
method="POST" style="display:inline;">
                            @csrf
                            @method('DELETE')
                            <button type="submit" class="delete-btn"
```

```
onclick="return confirm('Are you sure you want to delete this
task?');">Delete</button>
                        </form>
                    </div>
                </li>
            @empty
                <li>No tasks yet. Add one!</li>
            @endforelse
        </ul>
    </div>
</body>
</html>
```

12.4. Membuat View `edit.blade.php`

Buat file `resources/views/tasks/edit.blade.php` untuk formulir pengeditan tugas.

```
<!-- file: resources/views/tasks/edit.blade.php -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Edit Task</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        .container { max-width: 600px; margin: auto; background: #f9f9f9;
padding: 20px; border-radius: 8px; box-shadow: 0 0 10px rgba(0,0,0,0.1); }
        h1 { text-align: center; color: #333; }
        form div { margin-bottom: 15px; }
        form label { display: block; margin-bottom: 5px; font-weight: bold; }
        form input[type="text"], form input[type="checkbox"] { padding: 10px;
border: 1px solid #ddd; border-radius: 4px; width: calc(100% - 22px); }
        form input[type="checkbox"] { width: auto; margin-right: 10px; }
        form button { padding: 10px 15px; background: #28a745; color: white;
border: none; border-radius: 4px; cursor: pointer; }
        form button:hover { background: #218838; }
        .back-link { display: block; text-align: center; margin-top: 20px;
color: #007bff; text-decoration: none; }
        .back-link:hover { text-decoration: underline; }
        .error { color: red; font-size: 0.9em; margin-top: 5px; }
    </style>
</head>
<body>
    <div class="container">
        <h1>Edit Task</h1>

        <form action="{{ route('tasks.update', $task->id) }}" method="POST">
            @csrf
            @method('PUT')

            <div>
                <label for="description">Description:</label>
                <input type="text" name="description" id="description" value="
{{ old('description', $task->description) }}" required>
                @error('description')
                    <div class="error">{{ $message }}</div>
                @enderror
            </div>

            <div>
                <input type="checkbox" name="completed" id="completed" {{
$task->completed ? 'checked' : '' }}>
                <label for="completed">Completed</label>
            </div>

            <button type="submit">Update Task</button>
        </form>

        <a href="{{ route('tasks.index') }}" class="back-link">Back to Task
List</a>
    </div>
</body>
</html>
```

Dengan perubahan ini, aplikasi To-Do List Anda sekarang memiliki fungsionalitas CRUD yang lengkap. Anda dapat membuat, membaca, memperbarui, dan menghapus tugas.

---

13. Authentication and Authorization

Authentication adalah proses memverifikasi siapa pengguna tersebut, sedangkan *authorization* adalah proses memverifikasi apa yang diizinkan pengguna tersebut untuk lakukan. Laravel menyediakan solusi yang kuat dan mudah digunakan untuk kedua aspek ini.

13.1. Authentication

Laravel membuat implementasi *authentication* menjadi sangat sederhana. Hampir semua hal dikonfigurasi untuk Anda secara *out of the box*.

a. Menggunakan Laravel Breeze atau Jetstream

Cara termudah untuk memulai dengan *authentication* di Laravel adalah dengan menggunakan *starter kit* seperti Laravel Breeze atau Laravel Jetstream. Mereka menyediakan semua *scaffolding* yang diperlukan untuk *login*, *registration*, *password reset*, *email verification*, dan *password confirmation*.

Menginstal Laravel Breeze:

```
 composer require laravel/breeze --dev
php artisan breeze:install
php artisan migrate
npm install && npm run dev
```

Setelah instalasi, Anda akan memiliki *route* dan *view* untuk *authentication* yang sudah jadi. Anda bisa mengakses `/register` dan `/login` di aplikasi Anda.

b. Manual Authentication

Jika Anda tidak ingin menggunakan *starter kit*, Anda dapat mengelola *authentication* secara manual menggunakan fasad `Auth`.

Mencoba Login Pengguna:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(["email" => $`email, "password" => `$password])) {
    // Authentication passed...
    return redirect()->intended("dashboard");
}
```

Mengecek Apakah Pengguna Sudah Login:

```
if (Auth::check()) {
    // The user is logged in...
}

// Mengambil user yang sedang login
$user = Auth::user();
```

Logout Pengguna:

```
Auth::logout();

$request->session()->invalidate();

$request->session()->regenerateToken();

return redirect("/");
```

13.2. Authorization

Setelah pengguna diautentikasi, Anda perlu memverifikasi apakah mereka diizinkan untuk melakukan tindakan tertentu. Laravel menyediakan dua cara utama untuk mengelola *authorization*: *gates* dan *policies*.

a. Gates

Gates adalah *closure* sederhana yang menentukan apakah pengguna diizinkan untuk melakukan tindakan tertentu. Mereka sangat bagus untuk *authorization* yang sederhana dan tidak terkait dengan model tertentu.

Mendefinisikan Gate:

Gates biasanya didefinisikan di kelas `AuthServiceProvider` di metode `boot()`.

```
// file: app/Providers/AuthServiceProvider.php

use App\Models\User;
use Illuminate\Support\Facades\Gate;

public function boot(): void
{
    Gate::define(
        "update-post",
        function (User $`user, Post `$post) {
            return $`user->id === `$post->user_id;
        }
    );
}
```

Menggunakan Gate:

Anda dapat menggunakan *gate* di *controller* atau *view* Anda.

```
// Di Controller
if (Gate::allows("update-post", $post)) {
    // The user can update the post...
}

// Di Blade View
@can("update-post", $post)
    <!-- The current user can update the post -->
@endcan
```

b. Policies

Policies adalah kelas yang mengorganisir logika *authorization* untuk model atau sumber daya tertentu. Misalnya, jika aplikasi Anda membangun blog, Anda mungkin memiliki model `Post` dan *policy* `PostPolicy` untuk mengotorisasi tindakan pengguna seperti membuat atau memperbarui *post*.

Membuat Policy:

```
php artisan make:policy PostPolicy --model=Post
```

Ini akan membuat file `app/Policies/PostPolicy.php` dengan metode *stub* untuk setiap tindakan CRUD.

```php
// file: app/Policies/PostPolicy.php

<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Determine whether the user can view any models.
     */
    public function viewAny(User $user): bool
    {
        //
    }

    /**
     * Determine whether the user can view the model.
     */
    public function view(User $`user, Post `$post): bool
    {
        //
    }

    /**
     * Determine whether the user can create models.
     */
    public function create(User $user): bool
    {
        //
    }

    /**
     * Determine whether the user can update the model.
     */
    public function update(User $`user, Post `$post): bool
    {
        return $`user->id === `$post->user_id;
    }

    /**
     * Determine whether the user can delete the model.
     */
    public function delete(User $`user, Post `$post): bool
    {
        //
    }

    /**
     * Determine whether the user can restore the model.
     */
    public function restore(User $`user, Post `$post): bool
    {
        //
    }

    /**
     * Determine whether the user can permanently delete the model.
```

```
     */
    public function forceDelete(User $`user, Post `$post): bool
    {
        //
    }
}
```

Mendaftarkan Policy:

Anda perlu mendaftarkan *policy* di properti `$policies` dari `AuthServiceProvider`.

```
// file: app/Providers/AuthServiceProvider.php

protected $policies = [
    Post::class => PostPolicy::class,
];
```

Menggunakan Policy:

Anda dapat menggunakan *policy* di *controller* atau *view* Anda.

```
// Di Controller
public function update(Request $`request, Post `$post)
{
    $`this->authorize("update", `$post);

    // Update the post...
}

// Di Blade View
@can("update", $post)
    <!-- The current user can update the post -->
@endcan
```

Dengan *authentication* dan *authorization* Laravel, Anda dapat dengan mudah mengamankan aplikasi Anda dan mengontrol akses pengguna ke berbagai fitur dan sumber daya.

Referensi

- Laravel Documentation
- Composer
- Laravel Breeze
- Laravel Jetstream