Rapikan Controller

Pengantar

Controller memiliki peran besar dalam MVC(Model–View-Controller) project. Controller secara efektif sebagai penghubung yang mengambil request dari user, melakukan logic dan mengembalikan sebuah response. Jika kamu pernah mengerjakan proyek besar, kamu akan melihat di proyek tersebut memiliki banyak controller dan controller tersebut akan mulai menjadi berantakan dengan cepat tanpa anda sadari. Dibab ini kita akan melihat bagaimana merapikan controller yang bereantakan dalam laravel.

Untuk Pembaca

Bab ini ditujukan untuk siapa saja yang baru saja belajar laravel, yang mengerti dasar-dasar controller. Konsep yang dibahas relatif mudah untuk dipahami.

Masalah Controller yang Berantakan

Controller yang berantakan disebabkan oleh beberapa masalah bagi developer yaitu:

1. **Membuat controller sulit untuk melacak bagian tertentu dari kode atau kegunaannya.** Jika kamu ingin mengerjakan bagian tertentu dari sebuah kode yang ada di dalam controller yang berantakan, kamu akan menghabiskan waktu untuk mencari fungsi atau method dalam sebuah controller. Ketika kamu merapikan controller yang dipisahkan secara logic maka akan lebih mudah.
2. **Membuat controller sulit untuk menemukan bug berada secara akurat.** Seperti yang akan kita lihat dicontoh kode nanti, jika kita menangani authorization, validation, bisnis logic dan response dalam satu tempat, maka akan sulit untuk menemukan bug secara akurat.
3. **Membuat controller lebih sulit dalam menulis test untuk request yang kompleks.** Terkadang sulit menulis test untuk

method controller yang kompleks yang memiliki banyak baris kode dan melakukan banyak hal. Merapikan kode membuat testing lebih mudah. Kami akan membahas testing nanti dibab 6.

Controller yang Berantakan

Dibab ini,kita akan menggunakan sebuah contoh dari UserController:

```php
class UserController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        $this->authorize('create', User::class);

        $request->validate([
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ]);

        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Agar gambar diatas tetap bagus dan mudah dibaca, saya tidak menyertakan method `index()`, `edit()`, `update()`, dan `delete()` didalam controller. Tapi kita berasumsi bahwa method tersebut tetap ada dan kami juga menggunakan teknik dibawah ini untuk merapikan method tersebut juga. Untuk sebagian besar bab ini, kita akan fokus untuk merapikan atau mengoptimalkan method `store()`

Pindahkan Validation dan Authorization ke Form Request

Salah satu yang dapat kita lakukan adalah dengan memindahkan validation dan authorization ke class form request. Jadi, ayo kita lihat bagaimana kita bisa melakukan ini dicontroller untuk `store()` method.

Gunakan artisan command untuk membuat sebuah form request baru:

```
php artisan make:request StoreUserRequest
```

Command diatas akan membuat sebuah `app\Http\Request\StoreUserRequest` baru, seperti berikut:

```
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

kita akan menggunakan method `authorize()` untuk menentukan jika user diizinkan untuk melakukan request. Method tersebut akan mengembalikan true jika mereka diizinkan dan jika tidak maka

mengembalikan false. Kita bisa juga menggunakan method `rules()` untuk menentukan sebuah validation rules yang akan dijalankan pada request body. Kedua method akan dijalankan secara otomatis sebelum kita berhasil menjalankan sebuah kode didalam method controller tanpa perlu memanggil salah satu dari mereka secara manual. Jadi, ayo kita pindahkan authorization dari method controller store kedalam method `authorize()`. Setelah itu, kita bisa pindahkan validation rules dari controller ke method `rules()`. Kita juga sekarang mempunyai sebuah form request seperti ini:

```php
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ];
    }
}
```

controller kita seharusnya terlihat seperti ini:

```php
class UserController extends Controller
{
    public function store(StoreUserRequest $request): RedirectResponse
    {
        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Perhatikan bagaimana controller kita, kita mengubah argument pertama method `store()` dari `\Illuminate\Http\Request` ke `\App\Http\Requests\StoreUserRequest`. Kita juga mengurangi beberapa kode didalam method controller dengan memisahkannya ke dalam class request.

Catatan: Agar bekerja otomatis, kamu harus memastikan bahwa controller menggunakan traits `\Illuminate\Foundation\Auth\Access\AuthorizesRequests` dan `\Illuminate\Foundation\Validation\ValidatesRequests`.
Keduanya secara otomatis ditambahkan kedalam controller yang disediakan laravel saat instalasi baru. Jadi, jika controller kamu sudah extend ke controller (base), maka controller siap digunakan. Jika tidak maka pastikan kedua traits tersebut ada di dalam controller (base).

Pindahkan Common Login ke Actions atau Services

Langkah lain yang dapat kita lakukan untuk merapikan method store() pada controller yaitu memindahkan “bisnis logic” kita ke action atau service class.

Dalam kasus ini, kita bisa melihat bahwa fungsi utama dari method store() adalah membuat sebuah user, generate avatar user dan menugaskan kepada antrian job untuk mendaftarkan news letter user. Dalam pendapat pribadi saya, action lebih cocok untuk contoh ini dibandingkan service. Saya lebih memilih untuk menggunakan action untuk tugas kecil yang hanya melakukan satu hal tertentu. Sedangkan untuk kode yang lebih banyak berpotensi membuat panjangnya ratusan baris dan melakukan banyak hal, maka akan cocok untuk menggunakan service

Jadi, ayo kita buat action dengan membuat folder Actions baru didalam folder app kemudian buat sebuah class baru `StoreUserAction.php`. Kita dapat memindahkan kode kita ke action seperti berikut:

```php
class StoreUserAction
{
    public function execute(Request $request): void
    {
        $user = User::create([
            'name'     => $request->name,
            'email'    => $request->email,
            'password' => $request->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);
    }
}
```

Sekarang kita akan mengubah controller menggunakan action:

```
class UserController extends Controller
{
    public function store(
      StoreUserRequest $request,
      StoreUserAction $storeUserAction
    ): RedirectResponse {
        $storeUserAction->execute($request);

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Seperti yang anda lihat, sekarang kita dapat memindahkan bisnis logic keluar dari controller method ke dalam action. Ini sangat berguna, seperti saya sebutkan sebelumnya, controller pada dasarnya adalah penghubung untuk request dan response. Jadi, kita mengurangi yang berhubungan dengan apa yang kode lakukan dengan mebuat kode logic terpisah. Sebagai contoh, jika kita ingin mengecek authorization atau validation, kita akan langsung mengecek pada form request. Jika kita akan mengecek apa yang sedang dilakukan dengan request data, kita dapat mengecek pada action.

Manfaat besar lainnya untuk mengabstraksi kode menjadi terpisah berdasarkan class membuat testing lebih mudah dan cepat.

Menggunakan DTOs dengan Action

Manfaat besar lainnya dari memisahkan bisnis logic ke dalam service dan class adalah dapat menggunakan logic ditempat berbeda tanpa melakukan duplikasi kode. Contohya, ayo asumsikan bahwa kita memiliki sebuah `UserController` yang menangani tradisional web request dan sebuah `Api\UserController` yang menangani API request.

Untuk argument, kita dapat berasumsi bahwa struktur umum dari method `store()` untuk kedua controller akan sama. Tapi apa yang akan kita lakukan jika API request tidak menggunakan kolom `email`, tetapi menggunakan `email_address`? Kita tidak dapat melakukan pass request object ke class `StoreUserAction` karena class tersebut membutuhkan request object yang memiliki email.

Untuk solusi dari masalah tersebut, kita dapat menggunakan DTOs (data transfer object). Ini adalah cara yang sangat berguna untuk memisahkan data dan dapat menyebarkan ke seluruh proyek tanpa terikat apapun (dalam kasus ini adalah request).
Untuk menambahkan DTOs kedalam proyek kita, dapat menggunakan paket dari spatie `spatie/data-transfer-object` dan menggunakan command berikut untuk melakukan instalasi.

```
composer require spatie/data-transfer-object
```

Sekarang paket tersebut telah terinstall, Ayo buat folder DataTransferObject didalam folder app dan buat sebuah class `StoreUserDTO.php`. Kemudian pastikan bahwa class DTO yang kita buat melakukan extend terhadap `Spatie\DataTransferObject\DataTransferObject`. Kita kemudian bisa definsikan 3 propery seperti berikut.

```
class StoreUserDTO extends DataTransferObject
{
    public string $name;

    public string $email;

    public string $password;
}
```

Jika sudah selesai, kita dapat menambahkan method baru di `StoreUserRequest` sebelumnya ditambahkan dan return sebuah class `StoreUserDTO` seperti:

```php
class StoreUserRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'     => 'string|required|max:50',
            'email'    => 'email|required|unique:users',
            'password' => 'string|required|confirmed',
        ];
    }

    /**
     * Build and return a DTO.
     *
     * @return StoreUserDTO
     */
    public function toDTO(): StoreUserDTO
    {
        return new StoreUserDTO(
            name: $this->name,
            email: $this->email,
            password: $this->password,
        );
    }
}
```

Sekarang kita dapat melakukan update di controller untuk melakukan pass DTO ke class action:

```php
class UserController extends Controller
{
    public function store(
      StoreUserRequest $request,
      StoreUserAction $storeUserAction
    ): RedirectResponse {
        $storeUserAction->execute($request->toDTO());

        return redirect(route('users.index'));
    }

    public function unsubscribe(User $user): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Terakhir, kita harus melakukan update pada method class action untuk menerima DTO sebagai argument daripada request object:

```php
class StoreUserAction
{
    public function execute(StoreUserDTO $storeUserDTO): void
    {
        $user = User::create([
            'name'     => $storeUserDTO->name,
            'email'    => $storeUserDTO->email,
            'password' => $storeUserDTO->password,
        ]);

        $user->generateAvatar();
        $this->dispatch(RegisterUserToNewsletter::class);
    }
}
```

Hasil dari apa yang telah kita lakukan, sekarang kita telah berhasil memisahkan action dari request object. Ini artinya kita dapat menggunakan kembali action ini dalam tempat berbeda pada projek tanpa terikat terhadap spesifik struktur request. Kita sekrang akan bisa juga menggunakan pendekatan ini untuk CLI environment atau antrian job karena tidak terikat ke web request. Sebagai contoh, jika aplikasi kita memiliki fungsi untuk melakukan import user dari sebuah csv file, kita akan bisa membuat DTO dari csv data dan melakukan pass ke dalam action.

Kembali lagi kemasalah sebelumnya yaitu API request yang menggunakan `email_address` daripada `email`, kita sekarang akan dapat solusi dengan membuat DTO sederhana dan melakukan mengisi DTO `email` dengan request `email_address`. Sekarang bayangkan API request memiliki request class yang terpisah. Seperti contoh dibawah ini

```php
class StoreUserAPIRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return Gate::allows('create', User::class);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return [
            'name'          => 'string|required|max:50',
            'email_address' => 'email|required|unique:users',
            'password'      => 'string|required|confirmed',
        ];
    }

    /**
     * Build and return a DTO.
     *
     * @return StoreUserDTO
     */
    public function toDTO(): StoreUserDTO
    {
        return new StoreUserDTO(
            name: $this->name,
            email: $this->email_address,
            password: $this->password,
        );
    }
}
```

Menggunakan Resource atau Single-Use Controller

Cara terbaik untuk menjaga agar controller tetap rapi adalah pastikan bahwa keduanya adalah “Resource Controller” atau “Single-use Controller”. Sebelum kita melangkah lebih jauh dan mecoba mengubah contoh controller, Mari kita lihat apa arti dari keduanya.

Resource controller adalah sebuah controller yang menyediakan fungsionalitas berdasarkan resource tertentu. Jadi dalam kasus kita, resource kita adalah user Model dan kita ingin agar dapoat melakukan semua operasi CRUD(Create Read Update Delete) dalam model user. Sebuah resource controller umumnya mempunyai method

`index()`, `create()`, `store()`, `edit()`, `update()` dan `destroy()`. Tidak semua method perlu ada, tapi tidak akan memiliki method lain dalam list diatas. Dengan menggunakan jenis controller tersebut, kita bisa membuat routing lebih RESTful.

Single-use Controller adalah sebuah controller yang hanya memiliki satu method public `__invoke()`. Controller sangat berguna jika kamu memiliki sebuah controller yang methodnya tidak cocok dengan satu dari method RESTful yang kita miliki di resource controller.

Berdasarkan informasi diatas, kita dapat melihat bahwa `UserController` mungkin bisa diperbaiki dengan memindahkan method unsubscribe ke single-use controller.

Jadi ayo buat sebuah controller baru menggunakan artisan command:

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

Lihat bagaimana kita menambahkan `-i` pada command di atas yang akan membuat controller baru yang invokable, single-use. Kita sekarang memiliki sebuah controller seperti ini:

```
class UnsubscribeUserController extends Controller
{
    public function __invoke(Request $request)
    {
        //
    }
}
```

Sekarang kita dapat memindahkan kode method dan menghapus unsubscribe method dari controller lama:

```
class UnsubscribeUserController extends Controller
{
    public function __invoke(Request $request): RedirectResponse
    {
        $user->unsubscribeFromNewsletter();

        return redirect(route('users.index'));
    }
}
```

Pastikan bahwa anda ingat untuk mengubah route pada file `routes/web.php` gunakan `UnsubscribeController` daripada `UserController` untuk method ini.

Kesimpulan

Pada bab ini seharusnya memberikan kamu sebuah wawasan tentang berbagai cara yang dapat dilakukan untuk merapikan controller pada projek laravel. Harap diingat bahwa teknik yang digunakan disini adalah opini pribadi saya. Saya yakin bahwa ada developer lain yang akan menggunakan pendekatan yang benar-benar berbeda untuk membangun controller mereka. Bagian yang paling penting adalah tetap konsisten dan menggunakan pendekatan yang cocok dengan alur kerja anda (team).