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).

Tips Cepat & Mudah Untuk Mempercepat Aplikasi

Pengantar

Diperkirakan 40% orang akan meninggakan sebuah website jika membutukan waktu lebih lama dari 3 detik saat dimuat. Jadi dari sudut pandang bisnis sangat luar biasa penting untuk memastikan bahwa waktu memuat tetap di bawah 3 detik.

Oleh karena itu, Setiap kali saya menulis kode untuk proyek laravel, saya mencoba untuk memastikan kode optimal semaksimal mungkin dalam waktu yang diberikan serta kendala biaya. Jika saya pernah bekerja untuk proyek apapun, saya selalu mencoba untuk menggunakan teknik tersebut untuk memperbarui kode yang berjalan lambat untuk meningkatkan pengalaman user.

Dalam bab ini, kita akan melihat beberapa teknik yang saya gunakan (atau saran dari developer lain) untuk mempercepat peningkatan untuk klien dan web laravel serta aplikasi pribadi.

Untuk Pembaca

Bab ini ditujukan untuk laravel developer dalam tahap apapun yang sedang mencari beberapa tips cepat dan mudah untuk diterapkan pada aplikasi yang merekan bangun.

Hanya Ambil Field Yang Dibutukan di Query Database

Salah satu cara mempercepat laravel dengan mengurangi jumlah data yang ditarik antara aplikasi dan database. Cara yang dapat kamu lakukan dengan hanya spesifik terhadap kolom yang dibutuh dalam query dengan menggunakan select. Sebagai contoh, ibaratkan kamu memiliki sebuah User model yang mempunyai 20 kolom berbeda. Sekarang, bayangkan kamu memiliki 10,000 user dalam sistem dan

kamu mencoba melakukan beberapa proses setiap dari kolom tersebut. Contoh kodenya seperti berikut:

```
$users = User::all();

foreach ($users as $user) {
    // Do something here
}
```

Query diatas akan bertanggung jawab untuk mengambil 200,000 kolom data. Tapi, bayangkan ketika kamu memproses setiap user, kamu sebenarnya hanya menggunakan kolom `id`, `first_name` dan `last_name`. Jadi ini artinya dari 20 kolom, 17 diantaranya tidak digunakan. Jadi yang bisa kita lakukan adalah secara spesifik menentukan kolom yang diperlukan dalam query. Dalam kasus ini, kodenya terlihat seperti ini:

```
$users = User::select(['id', 'first_name', 'last_name'])->get();

foreach ($users as $user) {
    // Do something here
}
```

Dengan melakukan ini, kita mengurangi jumlah kolom yang dikembalikan dalam query dari 200,000 menjadi 30,000. Meskipun mungkin ini tidak akan terlalu berefek pada beban IO database, tapi akan mengurangi network traffic antara aplikasi dan database. Ini (mungkin) dikarenakan lebih sedikat data diserialize, dikirim dan deserialize dibandigkan jika mengambil semua kolom yang ada. Dengan mengurangi network traffic serta jumlah data yang diperlukan untuk diproses, akan membantu mempercepat web laravel. Harap dicatat, dalam contoh diatas kamu sebenarnya mungkin tidak pernah melakukan sesuatu seperti ini dan kamu akan menggunakan chunk atau pagination tergantung situasinya. Contoh diatas hanya hanya untuk menunjukan kemungkinan lebih mudah dalam mengimplementasi solusi.

Solusi ini mungkin tidak akan berpengaruh besar dalam sebuah web atau aplikasi kecil. Namun, ini adalah sesuatu yang dapat membantu mengurangi waktu muat dalam aplikasi dimana performa adalah hal penting yang harus dimiliki. Kamu mungkin juga melihat peningkatan yang bagus jika anda melakukan query ke sebuah table yang memiliki kolom BLOB atau TEXT. Kedua field tersebut dapat menyimpan data megabyte dan berpotensi meningkatkan lama waktu query. Jadi jika model tabel anda memiliki salah satu dari kolom tersebut, pertimbangkan secara spesifik tabel yang kamu butuhkan dalam query untuk mengurangi waktu load.

Gunakan Eager Loading Sebisa Mungkin

Ketika kamu menggunakan sebuah model untuk mengambil data dari database dan melakukan sebuah proses dari relasi model, sangat penting untuk menggunakan eager loading. Eager loading sangat sederhana di laravel dan pada dasarnya mencegah dalam menghadapi masalah N+1 pada data. Masalah ini disebabkan karena membuat query N+1 di database, dimana N adalah jumlah item yang diambil dari database. Untuk penjelasan yang lebih baik dan beberapa konteks, mari kita lihat contoh dibawah ini.

Bayangkan kamu memiliki dua model (Comment dan Author) dengan relasi 1-ke-1 diantara mereka. Sekarang bayangkan bahwa kamu memiliki 100 comment dan kamu ini melakukan loop setiap data dan mengeluarkan nama author.

Tanpa eager loading, kodenya akan seperti berikut:

```
$comments = Comment::all();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

Kode diatas akan menghasilkan 101 query database! Query pertama akan mengambil semua data comment. Dan seratus query lainnya akan

berasal dari mengeluarkan nama author pada setiap pengulangan loop. Jelas sekali, ini dapat menyebabkan masalah performa dan memperlambat aplikasi, jadi bagaimana kita memperbaiki masalah ini?

Dengan menggunakan eager loading, kita bisa mengubah kode menjadi:

```
$comments = Comment::with('authors')->get();

foreach ($comments as $comment) {
    print_r($comment->author->name);
}
```

Seperti yang kamu lihat, kode diatas terlihat sama dan masih dapat dibaca. Dengan menambahkan `::with('authors')` akan mengambil semua comment dan query lainnya untuk mengambil author hanya sekali. Jadi artinya kita memangkas query dari **101** menjadi **2**.

Bagaimana Memaksa Laravel Menggunakan Eager Loading

Dalam laravel, kamu dapat mencegah lazy loading. Fitur ini sangat berguna karena membantu untuk memastikan bahwa relasi menggunakan eager loading. Sebagai hasilnya, dapat meningkatkan performa dan mengurangi jumlah query yang dibuat database seperti contoh diatas.

Sangat sederhana untuk mencegah lazy load. Semua yang perlu kita lakukan adalah menambahkan kode dibawah ke method `boot()` pada `AppServiceProvider`:

```
Model::preventLazyLoading();
```

Jadi pada AppServiceProvider akan terlihat seperti berikut:

```php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

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

Mengizinkan Eager Loading Pada Production Environment

Anda mungkin hanya ingin mengaktifkan fitur ini saat mengembangkan aplikasi pada local environment. Dengan melakukan hal itu, kamu akan diperingatkan jika kode kamu terdapat sebuah lazy loading ketika membangun sebuah fitur baru, tapi tidak mengganggu web production. Untuk alasan ini method `preventLazyLoading()` menerima boolean sebagai argument, jadi kita bisa menggunakan baris berikut:

```php
Model::preventLazyLoading(! app()->isProduction());
```

Jadi, pada `AppServiceProvider`, akan terlihat seperti berikut:

```php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        // ...
        Model::preventLazyLoading(! app()->isProduction());
        // ...
    }
}
```

Dengan melakukan ini, fitur akan dimatikan jika APP_ENV adalah production jadi query lazy loading yang lolos tidak akan menyebabkan exceptions pada web anda.

Apa Yang Terjadi Ketika Menggunakan Lazy Load

Jika kita telah mengaktifkan fitur lazy load pada service provider dan kita mencoba melakukan lazy load terhadap relasi sebuah model, Sebuah exception `Illuminate\Database\LazyLoadingViolationException` akan ditampilkan.

Untuk memberikan sedikit konteks, ayo gunakan model kita `Comment` dan `Author` dari contoh atas. Katakanlah kita mengaktikan fitur tersebut.

Contoh kode yang akan mengembalikan exception:

```
$comments = Comment::all();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

Namun, kode dibawah ini tidak akan mengembalikan exception:

```
$comments = Comment::with('authors')->get();

foreach ($comments as $comment ) {
    print_r($comment->author->name);
}
```

Singkirkan Paket Yang Tidak Dibutukan dan Diinginkan

Buka composer.json file dan lihat setiap dependecies. Untuk setiap dependencies, tanya pada diri anda “apakah saya benar-benar membutuhkan paket ini?”. Jawaban anda sebagian besar adalah ya, tapi untuk sebagian mereka, mungkin tidak.

Setiap kali, kamu menambahkan sebuah composer library baru kedalam project anda, kamu berpotensi menambahkan extra code yang kemungkinan tidak perlu dijalankan. Laravel package biasanya terdapat service providers yang dijalankan pada setiap request yang melakukan register service dan menjalankan kode.

Jadi, katakanlah jika kamu menambahkan 20 laravel package kedalam aplikasi atau project, kemungkinan minimal 20 class akan di instansiasi dan dijalankan pada setiap request. Meskipun ini tidak akan berdampak besar terhadap performa untuk situs atau aplikasi dengan traffic kecil, kamu pasti dapat melihat perbedaannya pada aplikasi yang lebih besar.

Solusinya adalah menentukan apakah kamu membutuhkan sebuah paket itu. Mungkin kamu menggunakan sebuah paket yang menyediakan berbagai fitur tapi anda sebenarnya hanya menggunakan satu fitur kecil dari package tersebut. Tanya kediri anda “Bisakah saya menulis kode ini dan menghapus package tersebut” tentu saja karena kendala waktu, tidak selalu untuk menulis kode anda sendiri, karena anda harus menulis, melakukan test dan maintain kode. Paling tidak dengan menggunakan package, kamu dapat menfaatkan komunitas open-source untuk melakukan hal tersebut. Tapi jika sebuah paket tersebut sederhana dan dapat diganti dengan kode sendiri, maka pertimbangkan untuk menghapusnya.

Cache, Cache, Cache!

Laravel datang dengan berbagai caching methods. Ini dapat membuat web anda menjadi lebih cepat atau ketikan web anda sedang aktif tanpa perlu membuat perubahan kode.

Route Caching

Karena cara laravel berjalan,menjalakan framework dan melakukan parse pada file route setiap request yang dibuat. Ini membutuhkan membaca file, parsing the content dan mempelajari serta memahami bagaimana cara aplikasi anda digunakan. Jadi laravel menyedia

sebuah perintah yang bisa digunakan untuk membuat sebuah single route yang dapat di parse lebih cepat.

```
php artisan route:cache
```

Tolong perhatikan jika anda menggunakan perintah ini dan mengubah route anda, anda perlu menjalakan perintah berikut:

```
php artisan route:clear
```

Ini akan menghapus file cached route jadi route terbaru dapat di daftarkan. Mungkin bermanfaat jika menambahkan 2 command tersebut kedalam script deploy jika belum ditambahkan. Jika tidak menggunakan sebuah script deploy, kamu mungkin akan menemukan package Laravel Executor yang berguna untuk membantu dalam menjalankan deployment.

Config Caching

Serupa dengan route caching, setiap kali sebuah request dibuat, laravel menjalankan setiap config file dalam project anda dibaca dan diparse. Jadi untuk menghentikan agar setiap file tidak perlu ditangani. Anda bisa menjalankan perintah yang dapat membuat satu file config cached.

```
php artisan config:cache
```

Seperti route caching diatas, kamu perlu untuk menjalankan perintah berikut setiap kali mengubah .env atau config file

```
php artisan config:clear
```

Event dan View Caching

laravel juga menyediakan dua perintah lainnya yang dapat digunakan untuk melakukan cache views dan events sehinggan sudah dikompilasi dan siap ketika sebuah request dibuat. Untuk cache events dan views, kamu dapat menggunakan beberapa perintah:

```
php artisan event:cache
```

```
php artisan view:cache
```

Seperti perintah caching lainnya, kamu perlu ingat untuk menghancurkan cache setiap kali anda membuat perubahan dengan menjalankan perintah:

```
php artisan event:clear
```

```
php artisan view:clear
```

Dimasa lalu, Saya melihat banyak developer melakukan cache pada config di local development environment dan menghabiskan waktu mencari tau kenapa perbuahan .env tidak muncul. Jadi, saya merekomendasikan hanya melakukan caching pada config dan routes di live sistem agar anda tidak mengalami masalah yang sama.

Caching Query dan Value

Didalam kode laravel, kamu bisa melakukan cache pada items untuk meningkatkan performa website. Sebagai sebuah contoh, bayangkan kamu memiliki query berikut:

```
$users = DB::table('users')->get();
```

Untuk membuat caching dengan query ini, kamu harus merubah code menjadi seperti berikut:

```
$users = Cache::remember('users', 120, function () {
    return DB::table('users')->get();
});
```

Kode diatas menggunakan method `remember()`. Yang dilakukan pada dasarnya adalah mengecek jika cache mempunyai sebuah item dengan key users. Jika ada, akan mengembalikan value cache. Jika tidak ada dalam cache, maka hasil yang di kembalikan adalah query `DB::table('users')→get()` dan juga di cache. Dalam kasus ini, item akan dicache untuk 120 detik.

Cache data dan query seperti ini bisa menjadi sangat efectiv untuk mengurangi pemanggilan database, mengurangi run time dan meningkatkan performa. Namun, sangat penting untuk diingat bahwa kamu mungkin terkadang perlu untuk menghapus item dari cache jika tidak valid.

Menggunakan contoh diatas, bayangkan kita telah melakukan query users cache. Sekarang bayangkan bahwa sebuah user telah dibuat, diupdate atau dihapus. Hasil dari query cached tidak akan valid dan up to date. Untuk memperbaiki ini, kita harus menggunakan laravel model observers to menghapus item dari cache. Artinya jika selanjutnya mencoba dan mengambil $users variable, sebuah database query baru akan memberikan hasil yang up to date.

Menggunakan Versi PHP Terbaru

Dengan setiap versi php keluar, performa dan kecepatan telah ditingkatkan. Kinsta menjalankan banyak test across banyak versi PHP dan platform berbeda (Laravel, Wordpress, Drupal, Joomla) dan hasilnya bahawa PHP 8.0 memberikan peningkatan performa.

Tips ini mungkin sedikit sulit diterapkan jika dibandingkan dengan tips diatas karena kamu perlu melakukan audit pada kode dan memastikan kamu bisa secara aman mengupdate versi terbaru PHP. Sebagai catatan tambahan, menggunakan sebuah rangkain test otomatis dapat membantu anda dalam melakukan upgrade.

Menggunakan Queues

Tips ini mungkin akan memakan waktu lebih lama dibandingkan beberapa tips diatas yang menggunakan kode untuk diimplementasikan. Meskipun begitu, tips ini mungkin menjadi satu yang paling bermanfaat dalam hal user experience.

Salah satu cara yang bisa digunakan adalah mengurangi waktu performa dengan menggunakan laravel queues. Jika sebuah code yang berjalan di controller atau class dalam sebuah request dimana tidak diperlukan untuk response di web browser, kamu bisa menggunakan queue.

Untuk lebih mudah dimengerti, lihat contoh berikut:

```
class ContactController extends Controller
{
    /**
     * Store a new contact.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(ContactFormRequest $request)
    {
        $request->storeContactFormDetails();

        Mail::to('mail@ashallendesign.co.uk')->send(
          new ContactFormSubmission()
        );

        return response()->json(['success' => true]);
    }
}
```

Kode diatas, ketika method store() dipanggil akan melakukan penambahan kontak form detail di database, mengirim email ke sebuah alamat untuk menginformasikan mereka mengenai form kontak submission baru dan mengembalikan response json. Masalah pada kode diatas adalah user harus menunggu sampai email terkirim sebelum menerima response di web browser. Meskipun ini mungkin hanya beberapa detik, bisa berpotensi membuat pengunjung keluar.

Untuk menggunakan sistem queue, kita harus mengubah kode kita menjadi seperti berikut:

```
class ContactController extends Controller
{
    /**
     * Store a new contact.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(ContactFormRequest $request)
    {
        $request->storeContactFormDetails();

        dispatch(function () {
            Mail::to('mail@ashallendesign.co.uk')->send(
              new ContactFormSubmission()
            );
        })->afterResponse();

        return response()->json(['success' => true]);
    }
}
```

Kode diatas pada method store() sekarang akan menambahkan kontak form detail pada database, queue pengiriman email dan mengembalikan response. Sekali response dikembalikan ke web browser user, email akan di tambahkan di queue akan diproses. Dengan melakukan ini, artinya kita tidak perlu menunggu email terkirim sebelum mengembalikan response.

Cek dokumentasi laravel untuk informasi bagaimana mengatur queue untuk web laravel atau aplikasi.

Kesimpulan

Pada bab ini seharusnya dapat memberikan gambaran bagaimana cara mempercepat project laravel tanpa perlu melakukan refactor sepenuhnya. Tentu saja, ada lebih banyak hal yang dapat anda lakukan tapi ini yang biasa saya gunakan ketika saya ingin cepat dalam meningkatkan performa.

Bagaimana Membuat Function Helpers Sendiri

Pengantar

Function helpers bisa sangat berguna dalam project laravel. Mereka dapat membantu dalam menyederhanakan kode dalam project secara mudah dan clean. Laravel datang dengan banyak function helpers seperti `dd()`, `abort()` dan `session()`. Tapi ketika project mulai menjadi besar, kamu mungkin akan menambahkan function sendiri.

Pada bab ini kita akan melihat bagaimana membuat dan menggunakan custom function helper PHP milik kita sendiri pada project laravel.

Untuk Pembaca

Bab ini ditujukan untuk siapa saja yang memahami basic penggunaan laravel dan ingin merapikan kodenya menggunakan function yang dapat diakses di seluruh sistem.

Membuat Helpers Function

Untuk menambahkan helper function anda, kita mulai dengan membuat sebuah file PHP baru. Jadi, ayo buat sebuah file helpers.php di dalam folder app. Sebagai catatan, lokasi ini tergantung preferensi pribadi. Lokasi umum lainnya yang pernah saya lihat yaitu didalam file `app/Helpers/helpers.php`. Jadi terserah anda untuk memilih folder yang paling cocok bagi anda.

Sekarang setelah kita menaruh file kita, kita bisa menambahkan helper function kedalamnya, kita akan membuat sebuah function super simple yang menkonversi menit ke jam.

Ayo kita tambahkan fungsinya ke `helpers.php` seperti berikut:

```php
<?php

if (! function_exists('seconds_to_hours')) {
    function seconds_to_hours(int $seconds): float
    {
        return $seconds / 3600;
    }
}
```

Seperti yang kamu lihat pada contoh diatas, function tersebut sangat sederhana. Namun, satu yang mungkin kamu lihat adalah nama fungsi ditulis dalam snake case (`seconds_to_hours`) dibandingkan menggunakan camel case (`secondsToHours`) seperti yang biasanya kamu lihat pada nama method di sebuah class. Kamu tidak perlu menggunakan snake case untuk mendefinisikan nama fungsi, tapi kamu akan menemukan bahwa semua fungsi helper laravel di tulis dengan snake case. Jadi, saya kemungkinan akan menyarankan menggunakan format tersebut jadi dapat mengikuti standar dan format yang diharapkan. Ini sebenarnya terserah anda.

Hal lain yang mungkin anda lihat adalah kita membungkus nama fungsi kedalam if statement. Ini digunakan untuk menghentikan kira dari deklarasi fungsi dengan nama yang sama dan telah di register. Sebagai contoh, katakanlah kita menggunakan fungsi yang mempunyai nama `seconds_to_hours` dan telah di register, ini dapat menghentikan kita dari register fungsi dengan nama yang sama. Untuk kasus ini kita dapat mengubah nama fungsi yang kita buat untuk menghindari hal tersebut.

Juga penting untuk diingat ketika membuat function helper hanya digunakan sebagai helpers saja. Mereka tidak digunakan dalam bisnis logic, tapi untuk membantu merapikan kode anda. Tentu saja, kamu dapat menambahkan logic yang kompleks, tapi jika ini yang kamu lakukan, saya mungkin menyarankan bahwa kode akan cocok di tempat lain seperti service class, action class atau trait.

Mendaftarkan Helpers Function

Sekarang kita telah membuat helper file, kita perlu melakukan register sehingga dapat digunakan. Untuk melakukan ini, kita bisa update file composer.json jadi file kita akan di load pada runtime di setiap request dan bisa digunakan. Ini dapat terjadi karena laravel menambahkan composer class loader pada file `public/index.php`.

Di file `composer.json` seharusnya terlihat seperti berikut:

```
"autoload": {
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

Dibab ini, kita hanya perlu menambahkan beberapa baris agar composer mengetahui file yang ingin diload:

```
"files": [
    "app/helpers.php"
],
```

Pada autoload section file di `composer.json` akan terlihat seperti berikut:

```
"autoload": {
    "files": [
        "app/helpers.php"
    ],
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

Sekarang kita dapat update secara manual file composer.json, selanjutnya kita melakukan dump pada autoload file dengan perintah:

```
composer dump-autoload
```

Menggunakan Helpers Function

Selamat! Fungsi helper sekarang sudah dapat digunakan. Untuk menggunakannya, caranya sederhana:

```
seconds_to_hours(331);
```

Karena telah diregistrasi sebagai global function, artinya kamu dapat menggunakannya di berbagai tempat sperti controllers, service class dan bahkan helper function lainnya. Bagian yang saya sukai tentang helper function adalah dapat dignakan dalam blade views. Sebagai contoh, bayangkan kita memiliki `TimeServiceClass` yang memiliki method `secondsToHours()` yang berfungsi seperti fungsi helper yang baru kita buat. Jika kita ingin menggunakan service class dalam blade views, kita mungkin akan melakukan seperti berikut:

```
{{ \App\Services\TimeService::secondsToHours(331) }}
```

Seperti yang bisa dibayangkan, jika ini digunakan ditempat berbeda, kemungkinan dapat membuat view berantakan.

Menggunakan Banyak Helpers Function

Sekarang kita dapat melihat bagaimana kita melakukan register dan menggunakan fungsi helpers, kita akan melihat bagaimana kita bisa melangkah lebih jauh. Sepanjang waktu,project laravel akan bertambah besar, kamu mungkin akan menemukan bahwa kamu memiliki banyak jumlah helpers dalam satu file. Seperti yang dibayangkan,

file mulai terlihat tidak teroganisir. Jadi , kita ingin berpikir untuk memisahkan fungsi menjadi file terpisah.

Sebagai contoh, coba bayangkan kita memiliki banyak helpers difile `app/helpers.php`;beberapa terkait dengan uang, waktu dan user setting. Kita bisa mulai memisahkan fungsi tersebut menjadi file file terpisah sebagai contoh: `app\Helpers\money.php`, `app\Helpers\time.php` dan `app\Helpers\settings.php`. Artinya kamu sekarang bisa menghapus `app\helpers.php` karen kita tidak membutuhkannya sekarang.

Setelah itu, kita bisa mengupdate file `composer.json` berdasarkan 3 file baru sebelumnya:

```
"autoload": {
    "files": [
        "app/Helpers/money.php",
        "app/Helpers/settings.php",
        "app/Helpers/time.php",
    ],
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

Kita harus ingat untuk melakukan dump pada composer autoload sekali lagi dengan perintah

```
composer dump-autoload
```

Kamu sekarang bisa melanjukan menggunakan fungsi dan memiliki keuntungan memisahkan file yang dipisahkan secara logic.

Kesimpulan

Pada bab ini menunjukan bagaimana cara membuat dan register php helper function pada project laravel. Ingat untuk tidak menggunakan meraka dalam menjalankan bisnis logic yang kompleks dan lihat mereka untuk merapikan sedikit kode anda.

Menggunakan Interface Untuk Menulis Kode PHP Yang Baik

Pengantar

Dalam programming, sangat penting untuk membuat kode anda mudah dibaca, dimaintain, diextendable dan mudah ditest. Salah satu cara yang bisa digunakan dalam kode kita adalah menggunakan interface.

Untuk Pembaca

Dibandingkan dengan bab-bab sebelumnya, pada bab ini mungkin terlihat sedikit menakutkan pada awalnya. Bab ini ditujukan kepada developer yang memiliki mengerti basic dari konsep dan pewarisan PHP OOP (Object Oriented Programming). Jika kamu mengetahui bagaimana cara menggunakan pewarisan pada kode php, pada bab ini semoga bisa dimengerti.

Apa Itu Interface

pada dasarnya, interface hanya mendeskripsikan apa yang seharusnya sebuah class lakukan. Mereka bisa digunakan untuk memastikan bahwa setiap class yang mengimplementasi interface mendefinisikan public method didalamnya.

Interface **bisa** :

- Digunakan untuk mendefinisikan method public sebuah class
- Digunakan untuk mendefinisikan constant untuk sebuah class

Interface **tidak bisa**:

- Diinstansiasi
- Digunakan untuk mendefinisikan method private atau protected sebuah class
- Digunakan untuk mendefinisikan property sebuah class

Interface digunakan untuk mendefinisikan methods public dimana sebuah class harus tambahkan. Penting untuk diingat bahwa hanya

nama method yang didefinisikan dan tidak menambahkan body di methods (Seperti method pada umumnya). Ini dikarenakan interface hanya digunakan untuk komunikasi antar object, dibandingkan berkomunikasi dan berperilaku seperti class. Untuk memberikan sedikit konteks, contoh ini menampilkan contoh interface yang mendefinisikan beberapa methods public:

```
interface DownloadableReport
{
    public function getName(): string;

    public function getHeaders(): array;

    public function getData(): array;
}
```

Menurut php.net interface mempunyai 2 tujuan utama:

1. Untuk mengizinkan developer membuat object dari class berbeda yang mungkin digunakan secara bergantian karena mereka mengimplementasikan satu atau banyak interface yang sama. Contoh umumnya adalah banyak database melakukan acess ke service, multiple payment gateway atau strategi caching yang berbeda. Implementasi yang berbeda mungkin dapat tertukar tanpa membutuhkan perubahan ke kode yang mereka digunakan.
2. Untuk mengizinkan function atau method untuk menerima dan mengoperasikan dalam sebuah parameter yang sesuai dengan interface, yang tidak memperdulikan object mana yang akan digunakan atau bagaimana diimplementasikan. Beberapa interface seringkali dinamai seperti iterable, cacheable, renderable atau mendeskripsikan perilaku.

Menggunakan Interface Dalam PHP

Interface bisa menjadi tak ternilai dari bagian codebase OOP(Object Oriented Programming). Mereka mengizinkan kita untuk memisahkan kode kita dan meningkatkan perluasan sistem. Untuk memberikan contoh, silahkan lihat kode berikut:

```php
class BlogReport
{
    public function getName(): string
    {
        return 'Blog report';
    }
}
```

Seperti yang kamu lihat, kita telah mendefinisikan sebuah class dengan sebuah method yang mengembalikan sebuah string. Dengan melakukan ini, kita telah mendefinisikan perilaku dari method jadi kita bisa lihat bagaimana `getName()` mengembalikan string. Namun, katakanlah bahwa kita memanggil method di dalam sebuah class. Class lainnya tidak peduli bagaimana string tersebut di buat, mereka hanya peduli apa yang dikembalikan. Sebagai contoh, lihat bagaimana kita memanggil method ini dalam sebuah class:

```php
class ReportDownloadService
{
    public function downloadPDF(BlogReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

Meskipun kode diatas bekerja, ayo bayangkan kita ingin menambahkan fungsi untuk mendownload sebuah user report yang dibungkus didalam `UserReport` class. Tentu saja, kita tidak dapat menggunakan method yang sudah ada didalam `ReportDownloadService` karena kita memaksakan bahwa hanya class `BlogReport` yang bisa di passed. Jadi kita harus menamai method yang sudah ada dan menambahkan method baru seperti berikut:

```php
class ReportDownloadService
{
    public function downloadBlogReportPDF(BlogReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }

    public function downloadUsersReportPDF(UsersReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

Maskipun kamu sebenarnya tidak bisa melihatnya, ayo asumsikan method yang ada diatas digunakan untuk melakukan download. Kita bisa pindahkan kode yang sama kedalam methods tapi kita tetap mempunyai kode yang sama. Selain itu, kita akan memiliki beberapa point dari entry kedalam class yang menjalankan kode yang sama. Ini berpotensi untuk menambahkan kerja extra dimasa depan ketika mencoba menambahkan kode atau menambahkan test.

Sebagai contoh, ayo bayangkan bahwa kita membuat sebuah `AnalyticsReport` baru;kita sekarang perlu menambahkan sebuah methods baru `downloadAnalyticsReportPDF()` kedalam class. Kamu bisa melihat bagaimana file tersebut berkembang dengan cepat. Ini akan sempurna ketika menggunakan sebuah interface.

Ayo mulai membuat satu;kita membuat `DownloadableReport` dan mendefinisikan seperti berikut:

```php
interface DownloadableReport
{
    public function getName(): string;

    public function getHeaders(): array;

    public function getData(): array;
}
```

Kita sekarang bisa update `BlogReport` dan `UsersReport` untuk mengimplementasikan `DownloadbleReport` interface yang terlihat dicontoh diatas. Tapi sebagai catatan, Saya mempunyai tujuan menulis kode yang salah untuk `UsersReport` jadi kita bisa mendemontrasikan sesuatu!

```
class BlogReport implements DownloadableReport
{
    public function getName(): string
    {
        return 'Blog report';
    }

    public function getHeaders(): array
    {
        return ['The headers go here'];
    }

    public function getData(): array
    {
        return ['The data for the report is here.'];
    }
}
```

```
class UsersReport implements DownloadableReport
{
    public function getName()
    {
        return ['Users Report'];
    }

    public function getData(): string
    {
        return 'The data for the report is here.';
    }
}
```

Jika kita mencoba dan menjalankan kode kita, kita akan mendapatkan error dengan alasan berikut:

1. method `getHeaders()` tidak ada
2. method `getName()` tidak menambahkan return type yang mana terdefinisikan di interface

3. method `getData()` mendefinisikan sebuah return type, tapi tidak sama dengan yang ada di interface

Jadi ubah UsersReport supaya sama dengan implementasi dari `DownloadableReport` interface, kita bisa merubahnya menjadi seperti berikut:

```
class UsersReport implements DownloadableReport
{
    public function getName(): string
    {
        return 'Users Report';
    }

    public function getHeaders(): array
    {
       return [];
    }

    public function getData(): array
    {
        return ['The data for the report is here.'];
    }
}
```

Sekarang kita mempunyai report class yang mengimplementasi interface yang sama,, kita dapat mengupdate `ReportDownloadService` seperti berikut:

```
class ReportDownloadService
{
    public function downloadReportPDF(DownloadableReport $report)
    {
        $name = $report->getName();

        // Download the file here...
    }
}
```

Kita sekarang bisa melakukan pass ke dalam sebuah UsersReport atau BlogReport object kedalam `downloadReportPDF()` tanpa error. Ini karena kita tau sekarang method yang diperlukan ada di report class dan return data type sesuai.

Sebagai hasil dari passing ke dalam sebuah interface dibandingkan sebuah class, ini mengizinkan kita untuk memisahkan ReportDownloadService dan class report berdasarkan **apa** yang method lakukan, dibandingkan dengan **bagaimana** mereka melakukannya.

Jika kita ingin membuat sebuah `AnalyticReport` baru, kita bisa membuatnya dengan mengimplementasi interface yang sama dan kemudian ini akan mengizinkan kita untuk melakukan pass report object ke method yang sama `downloadReportPDF()` tanpa perlu menambahkan method baru. Ini khusunya bisa berguna jika kamu membuat package atau framework sendiri dan ingin memberikan developer kemampuan untuk membuat class mereka sendiri. Kamu dapat memberitahu mereka interface yang dapat diimplement dan mereka dapat membuat class mereka sendiri. Sebagai contoh, dilaravel kamu bisa membuat custom cache driver sendiri dengan mengimplementasikan `Illuminate\Contracts\Cache\Store` interface.

Selain menggunakan interface untuk meningkatkan kode, saya cenderung menyukai interface karena mereka dapat bertindak sebagai dokumentasi. Sebagai contoh jika saya mencoba mencari tahu apa yang sebuah class lakukan dan tidak bisa lakukan, saya cenderung melihat interface terlebih dahulu sebelum melihat sebuah class dan menggunakannya. Interface memberitahu semua method yang bisa dipanggil tanpa peduli bagaimana method berjalan dibalik layar.

Perlu dicatat untuk semua pembaca laravel developer bahwa kamu akan sering melihat istilah kontrak dan interface yang digunakan bergantian. Menurut dokumentasi laravel, “Laravel contract adalah sekumpulan interface yang mendefinisikan core service yang disediakan framework”. Jadi penting untuk diingat bahwa contrack adalah interface, tapi interface belum tentu contract. Biasanya, sebuah contract hanya sebuah interface yang disediakan framework. Untuk informasi lebih lanjut menggunakan contract saya memberikan dokumentasi untuk dibaca karena menguraikan apa mereka, bagaimana menggunakan dan kapan menggunakan mereka.

Kesimpulan

Bab ini harusnya memberikan gambaran singkat tentang apa itu interface, bagaimana mereka digunakan dalam PHP dan keuntungan menggunakan mereka.

Menggunakan Strategy Pattern

Pengantar

Dalam pengembangan software dan web, selalu penting untuk menulis kode yang dapat dimaintain dan extend. Solusi pertama yang anda buat dapat berubah seiring waktu berjalan. Jadi kamu perlu memastikan kode yang ditulis tidak akan banyak ditulis kembali atau direfactor dimasa mendatang.

Strategy pattern bisa digunakan untuk meningkatakan extend dari kode anda dan juga meningkatkan maintain dari waktu ke waktu.

Untuk Pembaca

Pada bab ini kemungkinan bab yang paling kompleks pada buku ini. Ditulis untuk laravel developer yang memahami bagaimana interface bekerja dan menggunakan mereka untuk memisahkan kode anda. Jika kamu telah membaca bab sebelumnya, kamu seharunya dapat mengerti konsep seluruhnya. Opini pribadi saya, cara terbaik untuk memahami konsep yang ada pada bab ini adalah mencobanya sendiri ketika (atau saat telah selesai) membaca.

Juga sangat disarankan anda mengerti menganai dependency injection dan bagaimana laravel service container bekerja.

Apa Itu Strategy Pattern

Refactoring guru mendefinisikan strategy pattern sebagai sebuah “behavioral design pattern yang memungkinkan anda mendefinisikan sebuah family dari algoritma, menaruh setiap dari mereka ke class terpisah dan membuat object mereka dapat dipertukarkan”. Kedengarannya sedikit menakutkan untuk pertama kali, tapi saya janji tidak seburuk yang kamu kira. Jika kamu ingin membaca lebih mengenai design pattern, saya sangat merekomendasikan refactoring

guru. Mereka melakukan penjelasan yang bagus mengenai streategy pattern secara mendalam dan structural pattern lainnya.

Strategy pattern adalah pada dasarnya sebuah pettern yang membantu kita dalam memisahkan kode kita dan membuat dapat diextend.

Menggunakan Strategy Pattern Pada Laravel

Sekarang kita telah mengetahui dasar mengenai strategy pattern, sekarang lihat bagaimana kita bisa menggunakan dalam project laravel kita.

Ayo bayangkan kita memiliki sebuah aplikasi laravel dimana user bisa digunakan untuk mengambil nilai tukar dan konvensi mata uang. Sekarang, katakanlah aplikasi kita menggunakan sebuah external API (exchangerratesapi.io) untuk mengambil konvensi data uang terbaru.

Kita akan membuat class beriteranksi dengan API:

```
class ExchangeRatesApiIO
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the exchangeratesapi.io API here.

        return $rate;
    }
}
```

Sekarang, ayo gunakan class ini dalam sebuah method controller jadi kita bisa mereturn konvensi data uang untuk mata uang tertentu. Kita akan menggunakan dependency injection untuk resolve class dari container.

```php
class RateController extends Controller
{
    public function __invoke(
      ExchangeRatesApiIO $exchangeRatesApiIO
    ): JsonResponse {
        $rate = $exchangeRatesApiIO->getRate(
            request()->from,
            request()->to,
        );

        return response()->json(['rate' => $rate]);
    }
}
```

Kode ini akan bekerja sesuai expektasi, tapi kita mengikat class `ExchangeRatesApiIO` ke method controller. Ini artinya jika kita memilih untuk melakukan migrasi untuk menggunakan API berbeda, seperti Fixer, di masa mendatang, kita perlu mengubah semuanya didalam codebase yang menggunakan class `ExchangeRatesApiIO` dengan class baru. Seperti yang dibayangkan, dalam project besar, ini bisa menjadi tugas yang lamabat dan membosankan. Jadi untuk menghidari masalah ini, dibandingkan mencoba melakukan inisialisasi sebuah class dalam method controller, kita menggunakan strategy pattern dalam melakukan bind dan resolve sebuah interface.

Ayo mulai dengan membuat sebuah interface baru `ExchangeRatesService`:

```php
interface ExchangeRatesService
{
    public function getRate(string $from, string $to): float;
}
```

Kita sekarang dapat mengubah class ExchangeRatesApiIO untuk mengimplementasikan interface diatas:

```php
class ExchangeRatesApiIO implements ExchangeRatesService
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the exchangeratesapi.io API here.

        return $rate;
    }
}
```

Sekarang kita telah selesai melakukan hal tersebut, kita bisa mengupdate method controller kita untuk melakukan inject terhadap interface dibandingkan class:

```php
class RateController extends Controller
{
    public function __invoke(
      ExchangeRatesService $exchangeRatesService
    ): JsonResponse {
        $rate = $exchangeRatesService->getRate(
            request()->from,
            request()->to,
        );

        return response()->json(['rate' => $rate]);
    }
}
```

Tentu saja, kita tidak bisa menginisialisasi sebuah interface;kita ingin menginisialisasi class `ExchangeRatesApiIO`. Jadi, kita perlu memberitahu laravel apa yang akan dilakukan ketika mencoba dan resolve interface dari container. Kita bisa melakukan ini dengan menggunakan sebuah service provider. Beberapa orang lebih menyukai untuk menyimpan hal ini didalam `AppServiceProvider` dan menyimpan semua binding di satu tempat. Namun, saya memilih membuat sebuah provider terpisah untuk setiap binding yang saya ingin buat. Ini murni karena preferensi pribadi dan apapun yang ada rasa cocok dengan alur kerja anda.

Ayo buat sebuah service provider baru menggunakan perintah artisan:

```
php artisan make:provider ExchangeRatesServiceProvider
```

Selanjutnya kita perlu ingat untuk register provider ke dalam `app\config.php` seperti berikut:

```
return [

    'providers' => [
        // ...
        \App\Providers\ExchangeRatesServiceProvider::class,
        // ...
    ],

];
```

Sekarang, kita bisa menambahkan kode kita ke service provider untuk melakukan bind interface dan class:

```
class ExchangeRatesServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(
          ExchangeRatesService::class,
          ExchangeRatesApiIO::class
        );
    }
}
```

Sekarang kita telah menyelesaikan semua ini, ketika kita melakukan dependency injection ke interface ExchangeRatesService didalam method controller kita, kita akan menerima sebuah class ExchangeRatesApiIO yang bisa kita gunakan.

Binding Multipe Class Ke Interface

Sekarang kita telah mengetahui bagaimana melakukan bind sebuah interface ke sebuah class, mari kita berpikir sedikit lebih jauh. Ayo bayangkan bahwa kita ingin bisa memilih menggunakan

ExchangeRatesApi.io atau Fixer.io kapanpun dengan hanya mengupdate sebuah kolom config.

Kita tidak memiliki class untuk Fixer.io, jadi ayo buat dan pastikan mengimplementasikan interface `ExchangeRatesService`:

```
class FixerIO implements ExchangeRatesService
{
    public function getRate(string $from, string $to): float
    {
        // Make a call to the Fixer API here and fetch the exchange rate.

        return $rate;
    }
}
```

Sekarang kita membuat sebuah kolom di file `config/services.php`:

```
return [

    //...

    'exchange-rates-driver' => env('EXCHANGE_RATES_DRIVER'),

];
```

Kita sekarang bisa mengupdate service provider untuk mengubah class mana yang akan dikembalikan kapanpun kita resolve interface dari controller:

```php
class ExchangeRatesServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(ExchangeRatesService::class, function ($app) {
            $driver = config('services.exchange-rates-driver');

            if ($driver === 'exchangeratesapiio') {
                return new ExchangeRatesApiIO();
            }

            if ($driver === 'fixerio') {
                return new FixerIO();
            }

            throw new Exception('The exchange rates driver is invalid.');
        });
    }
}
```

Sekarang jika kita memasang exchanges rates driver di `.env` menjadi `EXCHANGE_RATES_DRIVER=exchangeratesapiio` dan mencoba untuk resolve `ExchangeRatesService` dari controller, kita akan menerima sebuah class `ExchangeRatesApiIO`. Jika kita memasang exchanges rates driver di `.env` menjadi `EXCHANGE_RATES_DRIVER=fixerio` dan mencoba untuk resolve `ExchangeRatesService` dari controller, kita akan menerima sebuah class `FixerIO`. Jika kita memasang driver ke hal lain secara tidak sengaja, sebuah exception akan di berikan untuk memberitahu bahwa kita salah.

Berdasarkan fakta bahwa kedua class mengimplementasikan interface yang sama. Kita dapat dengan mudah mengubah kolom EXCHANGE_RATES_DRIVER di file .env dan tidak perlu mengubah kode lain dimanapun.

Kesimpulan

Apakah otakmu mulai terbakar? Jika iya, jangan khawatir! Saya pun, ketika menemukan topik ini sulit dipahami saat pertama kali belajar. Saya tidak berpikir untuk mulai benar-benar memahami sampai saya melakukan praktek dan menggunakannya. Jadi, nasihat

saya untuk menghabiskan sedikit waktu berexperimen dengan diri anda. Sekali kamu nyaman menggunakannya, saya jamin kamu akan mulai menggunakannya pada project anda sendiri.

Bab ini memberikan anda gambaran apa itu strategy pattern dan bagaimana menggunakannya dalam laravel untuk meningkatkan extendability dan maintain kode anda.

Buat Aplikasi Laravel Anda Mudah Di Test

Pengantar

Testing adalah bagian yang tidak terpisahkan dari development web dan software. Testing membantu untuk memberikan keyakinan bahwa kode yang anda tulis sesuai dengan kriteria dan juga mengurangi kemungkinan kode anda memiliki bug. Faktanya, TDD (test driven development) sebuah pendekatan development yang populer, sebenarnya berfokus pada test sebelum kode ditambahkan pada aplikasi sebenarnya.

Untuk Pembaca

Bab ini ditujukan untuk developer yang baru di dunia laravel tapi mengerti basic dari tests. Kita tidak akan membahas bagamana cara menulis basic tests, tapi akan menunjukan bagaimana pendekatan yang sedikit berbeda dari kode anda untuk meningkatkan kualitas kode dan kualitas tests.

Mengapa Harus Menulis Test?

Tests sering dianggap sebagai renungan dan sebuah “nice to have” untuk kode yang akan ditulis. Ini terlihat terutama pada organisasi dimana goals dan waktu memberikan tekanan pada tim developers. Dan sejujurnya, jika kamu hanya mencoba untuk mendapatkan MVP (Minimum Viable Product) atau sebuah prototype yang dibagun bersama secara cepat, mungkin test akan sedikit mengambil waktu. Tapi realitanya bahwa menulis test sebelum kode di rilis ke production adalah pilihan terbaik!

Ketika kamu menulis test, kamu melakukan banyak hal:

- **Menemukan bug lebih awal –** Jujur saja, berapa kali anda menulis kode, menjalankan sekali atau dua kali, dan kemudian melakukan commit ke version control system. Aku akan

mengangkat tangan, karena sudah melakukannya. Anda berpikir dalam diri anda “Kelihatannya sudah benar dan terlihat berjalan, saya yakin akan baik-baik saja”. Setiap saya melakukan hal ini, saya berakhir dengan dengan pull request pada github telah ditolak atau bug muncul pada production. Jadi dengan menulis test, kamu bisa menemukan bug sebelum melakukan commit pada pekerjaan anda dan sedikit rasa percaya diri setiap kali merilis ke production.

- **Membuat pekerjaan dan refactoring dimasa depan lebih mudah** – Bayangkan kamu perlu untuk melakukan refactor satu dari core class pada aplikasi anda. Atau mungkin perlu menambahkan kode baru pada class tersebut untuk melakukan extend functionality. Tanpa test, bagaimana kamu mengetahui dengan pasti bahwa mengubah atau menambahkan kode apapun tidak akan merusak kegunaan fungsi? Tanpa banyak manual test, tidak banyak cara untuk mengecek dengan cepat. Jadi dengan menulis test ketika kamu menulis versi pertama dari kode tersebut, kamu bisa memperlakukan mereka sebagai regression tests. Artinya setiap kali kamu mengupdate kode, kamu bisa menjalankan tests untuk memastikan semua bekerja. Kamu juga bisa menyimpan test yang ditambahkan setiap kali kamu menambahkan kode baru jadi kamu bisa yakin kode yang ditambahkan bekerja.\
- **Merubah cara pendekatan menulis kode** – Ketika saya pertama kali belajar mengenai testing dan mulai menulis test pertama (untuk laravel menggunakan PHPunit), Saya dengan cepat menyadari bahwa kode saya sulit untuk menulis test. Sulit untuk melakukan hal-hal seperti mocking class, mencegah panggilan ke third-party API dan membuat beberapa assertions. Untuk bisa menulis kode yang bisa ditest, kamu harus melihat struktur dari class dan method dari sudut pandang yang berbeda.

Menulis Controller Tests

Untuk menjelaskan bagaimana kita bisa membuat kode anda dapat ditest, kita akan menggunakan contoh sederhana. Tentu saja, ada cara berbeda dimana kamu bisa menulis kode dan mungkin sederhana tapi tidak masalah. Tapi, semoga, dapat membantu menjelaskan secara keseluruhan.

Ayo lihat contoh pada method controller:

```php
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(Request $request): JsonResponse
    {
        $service = NewsletterSubscriptionService();
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

Method diatas, yang kita asumsikan dipanggil jika kamu membuat sebuah request POST ke `/newsletter/subscriptions`, menerima sebuah parameter `email` yang kemudian di pass ke sebuah service. Kita bisa asumsikan bahwa service tersebut menangani semua layanan berbeda yang diperlukan untuk menyelesaikan user subscriptions ke newsletter.

Untuk test pada method controller diatas, kita bisa membuat test seperti berikut:

```php
class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test */
    public function success_response_is_returned()
    {
        $this->postJson('/newsletter/subscriptions', [
            'email' => 'mail@ashallendesign.co.uk',
        ])->assertExactJson([
            'success' => true,
        ]);
    }
}
```

Disana ada satu masalah yang mungkin kamu perhatikan di dalam test kita. Test tersebut sebenarnya tidak mengecek bahwa method service class handle() dipanggil! Jadi, jika secara tidak sengaja kami menghapus atau melakukan comment dibaris tersebut pada controller, kita tidak akan benar-benar tau.

Menulis Controller Test Yang Baik

Apa Masalahnya?

Satu dari masalah yang kita punya disini adalah tanpa menambahkan extra kode terdapat tanda atau log bahwa service class dipanggil, sulit bagi kita untuk mengecek jika sudah ditulis.

Tentu kita bisa menambahkan banyak assertions di dalam test controller untuk menguji bahwa service class kode semua telah dijalankan. Tapi dapat menyebabkan tumpang tindih pada kode anda. Untuk argumen, ayo bayangkan bahwa aplikasi laravel kita mengizinkan user untuk register dan kapanpun mereka register mereka secara otomatis terdaftar untuk newsletter. Sekarang, jika kita menulis test untuk controller ini juga mengecek bahwa semua service class dijalankan, kita memiliki 2 kode duplikasi dari kode test. Ini artinya jika kita melakukan update cara service class berjalan, kita juga harus melakukan update ke semua test. Sejujurnya, terkadang kamu ingin melakukan hal tersebut. Jika kamu menulis sebuah fitur test dan menjalankan assertions terhadap proses end-to-end, ini akan cocok. Namun, jika kamu mencoba untuk

menulis unit test dan hanya ingin mengecek controller, pendekatan ini tidak akan bekerja dengan baik.

Bagaimana Bisa Kita Memperbaiki Masalah Ini?

Untuk meningkatkan test yang kita punya, kita bisa menggunakan mocking, service container dan dependency injection. Saya tidak akan terlalu banyak membahas mengenai service container, tapi, saya merekomendasikan membacanya karena dapat sangat membantu dan sebuah bagian inti dari laravel.

Singkatnya (istilah sangat mendasar), service container mengelola dependency class dan mengizinkan kita untuk menggunakan class yang telah laravel sediakan. Untuk memahami apa yang saya maksud, silahkan lihat contoh berikut.

Untuk membuat contoh kode kita bisa lebih diuji, kita bisa instansiasi `NewsletterSubscriptionService` menggunakan dependency injection untuk melakukan resolve dari service container, seperti ini:

```php
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @param  NewsletterSubscriptionService $service
     * @return JsonResponse
     */
    public function store(
      Request $request,
      NewsletterSubscriptionService $service
    ): JsonResponse {
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

Kita telah selesai menambahkan class `NewsletterSubscriptionService` sebagai sebuah argument ke method `store()` karena laravel menginzinkan dependency injection di controller. Pada dasarnya apa yang dilakukan adalah memberitahu laravel ketika memanggil method ini adalah “Hei, saya juga ingin kamu memberikan sebuah `NewsletterSubscriptionService` !”. Laravel kemudian menjawab dan mengatakan “Oke, Saya akan ambil satu sekarang untuk kamu dari service container”.

Pada kasus ini, service kita tidak mempunyai sebuah argument constructor, jadi bagus dan simpel. Namun, jika kita ingin pass argument di constructor, kita berpotensi membuat sebuah service provider yang menangani data yang akan dipass kedalam class ketika pertama kali kita melakukan instansiasi.

Karena kita sekarang melakukan resolve dari container, kita bisa mengupdate test kita seperti berikut:

```
class NewsletterSubscriptionControllerTest extends TestCase
{
    /** @test */
    public function success_response_is_returned()
    {
        // Create the mock of the service class.
        $mock = Mockery::mock(NewsletterSubscriptionService::class)
            ->makePartial();

        // Set the mocked class' expectations.
        $mock->shouldReceive('handle')
            ->once()
            ->withArgs(['mail@ashallendesign'])
            ->andReturnNull();

        // Add this mock to the service container to take
        // the service class' place.
        app()->instance(NewsletterSubscriptionService::class, $mock);

        $this->postJson('/newsletter/subscriptions', [
            'email' => 'mail@ashallendesign.co.uk',
        ])->assertExactJson([
            'success' => true,
        ]);
    }
}
```

Sekarang, dalam test diatas, kita mulai menggunakan Mockery untuk membuat tiruan dari class service. Kita kemudian memberitahu service class dimana waktu test selesai dijalankan, kita mengharapkan bahwa methods `handle()` akan dipanggil sekali dan memiliki satu parameter yang berisi mail@ashallendesign.co.uk. Setelah melakukan hal tersebut, kita kemudia memberitahu laravel “Hei laravel, jika kamu membutuhkan untuk melakukan resolve sebuah `NewsletterSubscriptionService`, maka ini untuk kamu kembalikan”.

Ini artinya sekarang pada controller kita, parameter kedua tidak sebenarnya menggunakan service class, tapi digantikan sebuah tiruan class tersebut.

Jadi, ketika kita menjalankan test sekarang, kita melihat bahwa method `handle()` sebenarnya dipanggil. Sebagai hasilnya, jika kita menghapus dimana kode tersebut dipanggil atau menambahkan sebuah logic yang mungkin mencegah untuk dipanggil, test tersebut akan gagal karena Mockery mendeteksi bahwa method tersebut tidak dipanggil.

Bonus Tip Testing

Mungkin ada saatnya anda berada dalam sebuah class dan ternyata beberapa major refactoring tidak dapat melakukan inject terhadap class anda (yang ingin anda lakukan mock) dengan pass sebagai tambahan argument pada method. Dalam kasus tersebut, kamu bisa menggunakan method helper resolve() yang disediakan laravel.

Method resolve() sederhananya mengembalikan sebuah class dari service container. Sebagai contoh kecil, ayo lihat bagaimana kita bisa mengupdate contoh method controller agar bisa ditest dengan Mockery tapi tanpa menambahkan argument tambahan:

```php
use App\Services\NewsletterSubscriptionService;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\Request;

class NewsletterSubscriptionController extends Controller
{
    /**
     * Store a new newsletter subscriber.
     *
     * @param  Request  $request
     * @return JsonResponse
     */
    public function store(Request $request): JsonResponse
    {
        $service = resolve(NewsletterSubscriptionService::class);
        $service->handle($request->email);

        return response()->json(['success' => true]);
    }
}
```

Kesimpulan

Pada bab ini seharunya memberikan sedikit wawasan bagaimana kamu bisa membuat aplikasi laravel dapat lebih di test dengan menggunakan service container, mocking dan dependency injection.

Ingat bahwa test adalah teman anda dan bisa menyelamatkan waktu anda, stres dan tekanan jika mereka menambahkan ketika kodeditulus pertama kali. Dan sebagai bonus, kualitas test yang baik dan meningkatkan test coverage biasanya (tapi tidak selalu) menyebabkan sedikit bug, yang artinya dukungan lebih sedikit dan client senang!