

# Building APIs using Laravel

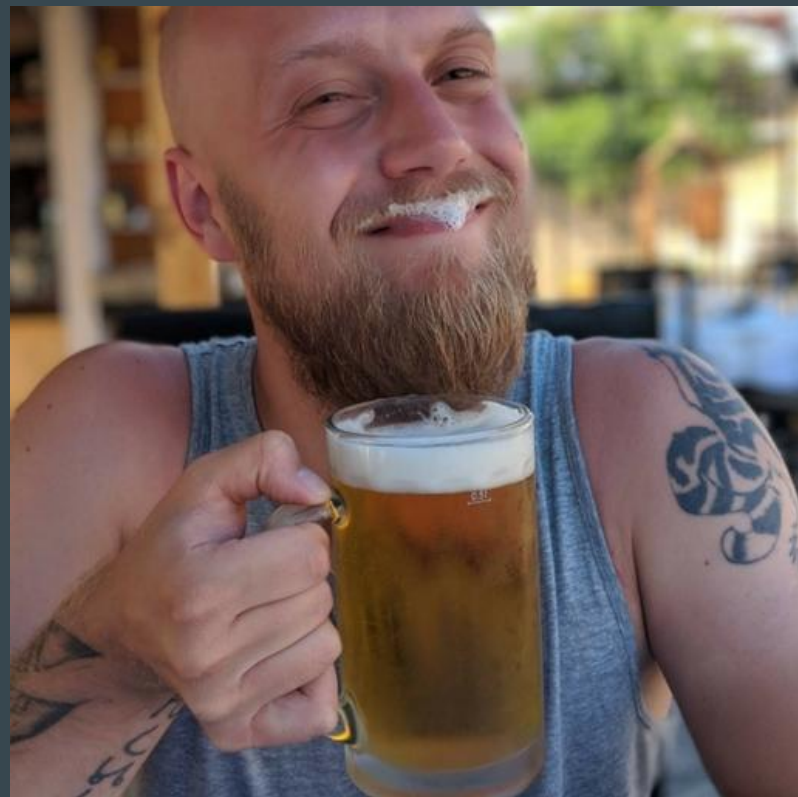
...

A simple approach to scale

# Oi, who are you?

I'm Steve McDougall (JustSteveKing)

- PHP User Group Organiser
- Conference Organiser
- PHP Advocate
- Bearded man



**How do people go wrong with APIs?**


# We have all hit that point of no return

- 100s of Models in our app directory
- 100s of Models in our app/Models directory
- More scopes than a game of Call of Duty

What if there was a better way?

# Setting up domains

Using a simple domain approach means you can split up your business logic easily.



```
app/  
- Domain  
  - Shared  
    - Builders  
      - UserBuilder.php  
    - Collections  
      - UserCollection.php  
    - Models  
      - User.php
```

# How do our Models look now?



```
namespace App\Domain\Shared\Models;

class User
{
    public function newEloquentBuilder($query)
    {
        return new UserBuilder($query);
    }

    public function newCollection(array $models = [])
    {
        return new UserCollection($models);
    }
}
```

# What do we add to our Builder classes?



```
namespace App\Domain\Shared\Builders;

use Illuminate\Database\Eloquent\Builder;

class UserBuilder extends Builder
{
    public function verifiedEmail(): self
    {
        return $this->whereNotNull('email_verified_at');
    }
}
```

# RouteServiceProvider modification



# We are used to seeing this

```
class RouteServiceProvider extends ServiceProvider
{
    protected $namespace = 'App\Http\Controllers';

    public const HOME = '/home';

    public function boot()
    {
        parent::boot();
    }

    public function map()
    {
        $this->mapApiRoutes();

        $this->mapWebRoutes();
    }

    protected function mapWebRoutes()
    {
        Route::middleware('web')
            ->namespace($this->namespace)
            ->group(base_path('routes/web.php'));
    }

    protected function mapApiRoutes()
    {
        Route::prefix('api')
            ->middleware('api')
            ->namespace($this->namespace)
            ->group(base_path('routes/api.php'));
    }
}
```

# Here is my version

```
class RouteServiceProvider extends ServiceProvider
{
    public function boot()
    {
        parent::boot();
    }

    public function map()
    {
        $this->mapRoutes();
    }

    protected function mapRoutes()
    {
        // Version 1
        Route::prefix('v1')
            ->middleware('api')
            ->group(
                base_path('routes/api/v1.php')
            );

        // Version 2

        // Version 3
    }
}
```

# JSON:API and Middleware

JSON:API has been properly registered with the IANA. Its media type designation is `application/vnd.api+json`.



```
class ContentTypeMiddleware
{
    public function handle(Request $request, Closure $next)
    {
        $response = $next($request);

        $name = Str::slug(config('app.name', 'api'), '.');
        $response->header(
            'Content-Type',
            "application/vnd.{$name}+json"
        );

        return $response;
    }
}
```

A simple implementation

# What is that about?

Here is an example:

*application/vnd.github.v3+json*

We build up our content type by:

*type "/" "vnd." subtype ["+" suffix] \*[";" parameter]*

# Handling Routing



```
// Typical Laravel Routing
Route::get('resources', 'ResourceController@index');

// Using the API Resource from Laravel
Route::apiResource('resources', 'ResourceController');

// Invokable Controllers
Route::get('resources', \App\Http\Controllers\IndexController::class);
```

There are quite a few ways to handle routing in Laravel





```
/**
 * Posts Resource
 */
Route::prefix('posts')->as('posts.')->group(function () {

    Route::get('/', \App\Http\Actions\V1\Posts\CollectionAction::class)->name('collection');

    Route::post('/', \App\Http\Actions\V1\Posts\CreateAction::class)->name('create');

    Route::get('{post}', \App\Http\Actions\V1\Posts\GetAction::class)->name('get');

    Route::patch('{post}', \App\Http\Actions\V1\Posts\UpdateAction::class)->name('update');

    Route::delete('{post}', \App\Http\Actions\V1\Posts\DeleteAction::class)->name('delete');

});
```

My typical approach

**Let's handle a route**



```
namespace App\Http\Actions\V1\Posts;

class CollectionAction
{
    public function __invoke(Request $request): Response
    {
        $posts = QueryBuilder::for(Post::class)
            ->allowedFilter('id', 'title', 'content', 'author.slug', 'category.slug')
            ->allowedIncludes('author', 'category')
            ->allowedSort('id', 'category.name', 'author.name', 'created_at')
            ->published()
            ->paginate();

        return response()->json(
            PostResource::collection($posts),
            Response::HTTP_OK
        );
    }
}
```

A simple and flexible Action to handle a stateless request.



```
namespace App\Http\Resources\V1\Posts;

class PostResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
            'id' => $this->id,
            'title' => $this->title,
            'content' => $this->content,
            'created' => [
                'string' => $this->created_at->toDateString(),
                'human' => $this->created_at->diffForHumans()
            ],
            'published' => $this->published,
            'author' => new AuthorResource($this->whenLoaded('author')),
            'categories' => CategoryResource::collection($this->whenLoaded('categories'))
        ];
    }
}
```

A clean and simple Resource

```
namespace App\Http\Resources\V1\Posts;

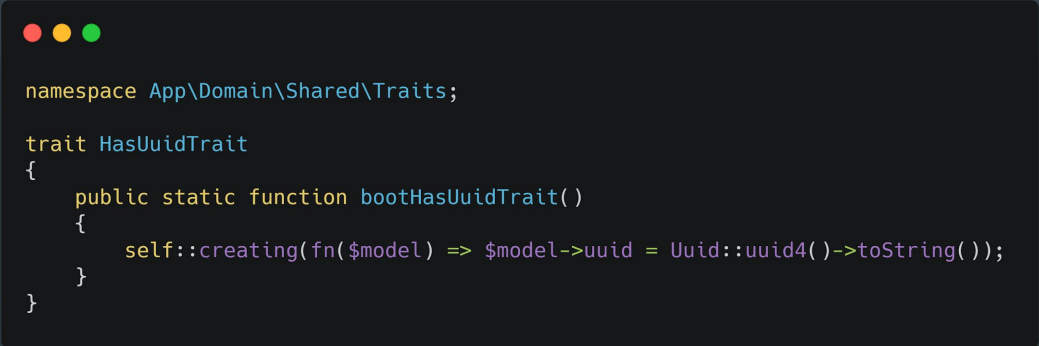
class PostResource extends JsonResource
{
    public function toArray($request): array
    {
        return [
            'type' => 'posts',
            'id' => $this->uuid,
            'attributes' => [
                'id' => $this->uuid,
                'title' => $this->title,
                'content' => $this->content,
                'created' => [
                    'string' => $this->created_at->toDateString(),
                    'human' => $this->created_at->diffForHumans()
                ],
                'published' => $this->published,
            ],
            'relationships' => [
                'author' => new AuthorResource($this->whenLoaded('author')),
                'categories' => CategoryResource::collection($this->whenLoaded('categories'))
            ],
            'links' => [
                'self' => route('posts.collection')
            ]
        ];
    }
}
```

We could be a little more advanced

# Handling Traits

Traits in Laravel are really handy, they allow you to share behaviour between objects to keep your code DRY.

Here is one I use quite a lot on my Models:



```
namespace App\Domain\Shared\Traits;

trait HasUuidTrait
{
    public static function bootHasUuidTrait()
    {
        self::creating(fn($model) => $model->uuid = Uuid::uuid4()->toString());
    }
}
```

# Observing behaviour in our API



# Model Observers are powerful

Model Observers in Eloquent are a powerful tool when used right. Sometimes you need to share this sort of behaviour which is why in the previous example I used a Trait - however consider the following scenario:

When an author creates a new post, he want this to automatically publish this post onto social media channels.

# A simple approach

- Register a PostCreated event
- Register Listeners for each social media you wish to publish to
- On each listener post through the API to the social media channel.

In a small scale application this would be fine. In a larger scale this is going to cause issues.

# A slightly better approach

- A Model Observer handles the created event and dispatches a post to social media Job.
- The post to social media job handles posting to each social media API.
- Alternatively the Model Observer could/should dispatch a job to handle posting to each social media API separately, meaning several Jobs are dispatched.

Again, this is a step forward. Posting to 3rd party APIs should be done as a background task - so we do not delay the response returning from our API.

# A more advanced approach

- Our author can select per post which channels we want to share this post on, as not all social media channels are equal.
- Our Model Observer will dispatch a Job for each of the selected channel.
- When our post has been published to each channel, we can update our Post Channel relationship with meta information such as when it was posted and even a reference ID to pull analytics from the API.

Not all systems will need this fine grained analytical data - but by taking a step back in our planning we can easily see how we can refactor towards it.



```
namespace App\Publishing\Jobs\V1\Posts;

class PublishToTwitter implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    protected Post $post;

    protected Author $author;

    public function __construct(Post $post, Author $author)
    {
        $this->post = $post;
        $this->author = $author;
    }

    public function handle(TwitterService $twitter, TweetBuilder $builder)
    {
        try {
            $twitter->post(
                $builder->build($this->post, $this->author)
            );
        } catch (Exception $e) {
            Log::error($e->getMessage());
        }
    }
}
```

## An Example Job

**A hat tip to TDD**

# Laravel and API testing

Laravel has a fantastic suite of testing tools, and you should be using them.

The latest release of 7.\* even has some great helpers for testing with a Http Client against 3rd party APIs.

If you aren't already, start adding tests - you will thank me in the long run.



```
namespace Tests\Feature;

class PostEndpointTest extends TestCase
{
    use DatabaseMigrations;

    public function testCollectionResourceEndpointStatus()
    {
        $response = $this->get('/v1/posts');

        $response->assertStatus(Response::HTTP_OK);
    }
}
```

A simple test



# What have we covered?

- Cleaner Models
- Cleaning up Routing Loading for an API
- Route Declarations, and there many forms
- Content Type middleware and JSON:API
- Actioning a Request coming into our API
- API Resources for consistency
- How useful Traits can be
- Mode Observers and examples of when to use them
- Dispatchable Jobs
- Testing is important, even if they are simple.

**APIs do not have to be hard. But they do have to be stateless.**

# Thanks for listening

Twitter: @JustSteveKing

GitHub: JustSteveKing