Introduction

This short PDF is about the fundamentals. It tries to show you how you can write robust, high-quality APIs in Laravel where you have a standard way for:

- Filters
- Sorting
- Including relationships
- Sparse fieldsets
- API Versioning
- Test-Driven Development

Achieving this requires a lot of code, but fortunately there are some packages and standards that can help us:

- JSON API
- laravel-query-builder by Spatie
- json-api by Tim MacDonald

👋 Hi! I'm Martin Joo. I tweet about Laravel every day, publish blog posts, and also wrote a book. You can find me here.

JSON API

We write so many APIs and if you think about it each of them is different:

- Different data structures.
- Relationships are included differently.
    - In some APIs, a relationship is included with other attributes.
    - In some APIs, there's a separate `relations` key for them.
    - In some APIs you don't get relationships at all, you have to call another endpoint.
    - Some APIs use `camelCase` while others use `snake_case` .
- The filtering and the sorting are different every damn time.

Wouldn't be great if there were a standard that generalizes all of these concepts? Yes, there is! And it's called JSON API.

JSON API is a specification that tells you how to structure your response data, how to implement filtering, sorting, relationship including in your request URLs.

Responses

Here's the basic structure of a JSON API response:

```
{
  "data": {
    "id": 1,
    "type": "threads",
    "attributes": {
      "slug": "this-is-a-thread",
      "title": "This is a thread",
      "body": "This is the actual content of the thread",
      "counters": {
        "share": 88,
        "like": 367,
        "reply": 6
      }
    },
    "relationships": {
      "author": {
        "data": { "type": "users", "id": 1}
      },
      "tags": [
        {"data": { "type": "tags", "id": 2}},
        {"data": { "type": "tags", "id": 16}},
      ]
    },
    "included": [
      {
        "id": 1,
        "type": "users",
        "attributes": {
          "name": "Micheal Scott",
          "email": "micheal.scott@dunder-mifflin.com"
        }
      },
      {
```

```
        "id": 2,
        "type": "tags",
        "attributes": {
          "name": "sales"
        }
      },
      {
        "id": 16,
        "type": "tags",
        "attributes": {
          "name": "paper"
        }
      }
    ]
  }
}
```

Some of the key things:

- Every resource has an ID and a type as a root level attribute
- Every other attribute lives in the attributes object
- The relationships key only contains some meta information about the relationships
- The included key actually contains the loaded relationships like author or tags

There are a tons of other thing in the actual specification, but I want to focus on the basics. That's the most important stuff. The relationship part can be a bit confusing, so let's clear this up.

In web development we have two main choices when it comes to relationships in APIs:

- Include (almost) everything and return a huge response.
- Include (almost) nothing and provide links to the clients.

There are pros and cons to both of these approaches and most of the time it depends on your application. But there is two good rules of thumbs:

- If you include (almost) everything you may end up with huge responses and N+1 query problems. For example, you return a response with 50 articles and each article loads the author. You end up with 51 queries.
- If you include (almost) nothing you have small responses but you may have too many requests to the server. For example, you return a response with 50 articles, after that the client makes 50 additional HTTP requests to get the authors.

JSON API supports both of these options:

```
{
  "relationships": {
    "author": {
      "links": {
        "self": "http://example.com/articles/1/relationships/author",
        "related": "http://example.com/articles/1/author"
      }
    },
    "comments": [{
      "data": {"type": "comments", "id": 5}
    }]
  },
}
```

In this example, the comments relationship is loaded while the author only contains a link where you can fetch the author. In fact, it contains two links and it's a little bit confusing so let's make it clear:

- related: with this URL I can manipulate the actual user (author). So if I make a `DELETE http://example.com/articles/1/author` request I delete the user from the users table.
- self: this URL only manipulates the relationship between the author and the article. So if I make a `DELETE http://example.com/articles/1/relationships/author` request I only destroy the relationship between the user and the article. But it won't delete the user from the users table.

As you can see in the comments key there is an ID and a type, but no attributes. Where is it? It lives in the included key:

```
"included": [{
  "type": "comments",
  "id": "5",
  "attributes": {
    "body": "First!"
  },
  "links": {
    "self": "http://example.com/comments/5"
  }
}]
```

This is a little bit confusing, so let's make it clear:

- Relationships: it only contains information about the relationships. It tells you that an article has author and comments relationships.
- Included: it contains the loaded relationships. In the example above the API endpoint only loads the comments, so in the included key you can find the comments but not the author. The author can be fetched using the links in the relationships object.

This is the part I like the least about JSON APi. In my opinion, it would be much better if there's only a relationships key that looks like this:

```
{
  "relationships": {
    "author": {
      "links": {
        "self": "http://example.com/articles/1/relationships/author",
        "related": "http://example.com/articles/1/author"
      }
    },
    "comments": [{
      "type": "comments",
      "id": "5",
      "attributes": {
        "body": "First!"
      },
      "links": {
        "self": "http://example.com/comments/5"
      }
    }]
  },
}
```

So it actually contains the comments and provides links for the author. However in this book, I will follow the official specification, but here's my opinion: you don't have to obey a standard just because it's a standard. You can use only the good parts from it that fit your application.

Either way, you have a reasonable default standard in your company / projects.

Laravel resources

Creating this response structure by hand can be exhausted. But fortunately there is a package called `json-api` by Tim MacDonald.

By using this package you need extend the `JsonApiResource` class in your resources instead of the `JsonResource` provided by Laravel. A resource class looks like this:

```
use TiMacDonald\JsonApi\JsonApiResource;

class CategoryResource extends JsonApiResource
{
  public function toAttributes($request): array
  {
    return [
      'name' => $this->name,
      'slug' => $this->slug,
    ];
  }
}
```

With this package you don't need to define:

- ID
- Type
- Attributes

It will produce this JSON automatically. By default it uses the `id` column from your models. You can override this behavior in the `AppServiceProvider`:

```
public function boot()
{
  JsonApiResource::resolveIdUsing(fn (Model $model) => $model->uuid);
}
```

In this example I use the field `uuid` as the `id` in the responses. You can also define relationships with the `toRelationships` method:

```php
use TiMacDonald\JsonApi\JsonApiResource;

class ThreadResource extends JsonApiResource
{
  public function toAttributes($request): array
  {
    return [
      'slug' => $this->slug,
      'title' => $this->title,
      'body' => $this->body,
      'counters' => [
        'share' => $this->share_count,
        'like' => $this->like_count,
        'reply' => $this->reply_count,
      ],
    ];
  }

  public function toRelationships($request): array
  {
    return [
      'author' => fn () => new AuthorResource($this->author),
      'category' => fn () => new CategoryResource($this->category),
      'tags' => fn () => TagResource::collection($this->tags),
    ];
  }
}
```

This will populate the *relationships* and the *included* keys as well. The important thing is that all array keys like *author* is a callback. These keys are lazily evaluated. It means that this package checks the *included* URL parameter in the request and if a relationship is included only then the callback will be executed. In this way the client is in perfect control about what relationships it needs, and only query the necessary ones.

Requests

JSON API also specifies how you the request URLs should look like. This includes:

- Filters
- Sorting
- Sparse fieldset
- Including relationships

Filtering

```
GET /threads?filter[title]=laravel&filter[title]=php
```

Here we say: "I only want threads that contains Laraval **and** PHP in the title". You can specify OR relation:

```
GET /threads?filter[name]=laravel,php
```

Sorting

```
GET /threads?sort=-like_count
```

Sorting threads descending based on like count. What about ascending?

```
GET /threads?sort=like_count
```

We just remove the - symbol. You can specify multiple columns:

```
GET /threads?sort=like_count,share_count
```

Including relationships

It's a very good practice to let the client decide when to load a relationship and when not to. Later we can leverage Laravel API Resources to accomplish this. With JSON API it's really simple:

```
GET /threads?include=tags
```

A thread can have a lot of tags, and maybe we don't want to include them every single time, so we let the client decide it. We can include multiple stuff:

```
GET /threads?include=tags,author
```

Sparse fieldset

SELECT * queries can be slow and time-consuming, and often we don't need every single column in the response. We can solve it like this:

```
GET /threads?fields[threads]=id,title,body
```

We can also specify fields for included relationships:

```
GET /threads?include=author&fields[author]=id,name
```

And of course JSON API also specifies pagination, but that's already solved by Laravel out of the box, so we're gonna skip this part. The next question is: how do we implement all this complicated, generic stuff for every model? Luckily we don't have to. We already have the solution. Let's learn about Query Builders!

Spatie QueryBuilder

In my opinion this is the most important package when building great APIs. This package implements all of the:

- Sorting
- Filtering
- Including relationships
- Spare fieldsets
- Paginating

These problems are solved out of the box using QueryBuilder classes. Let's see how it looks like!

Filtering

```
// GET /threads?filter[title]=laravel&filter[title]=php
$threads = QueryBuilder::for(Thread::class)
  ->allowedFilters(['title', 'body'])
  ->get();
```

We create a QueryBuilder for a specific model (in this case Thread), and define what filters we want to allow. Then it will get these filters from the request, and apply them by where filters. That's it! You're done. You have a fully functioning filter. You can also apply additional where statements or scopes:

```
$threads = QueryBuilder::for(Thread::class)
  ->allowedFilters(['title', 'body', 'author.name'])
  ->where('category_id', $category->id)
  ->wherePublic()
  ->get();
```

Sorting

```
// GET /threads?sort=reply_count
$threads = QueryBuilder::for(Thread::class)
  ->allowedSorts(['like_count', 'reply_count'])
  ->get();
```

We can specify what columns can be used in the ORDER BY clause. If the URL doesn't contain a sort query, we can apply a default sort:

```
// GET /threads
$threads = QueryBuilder::for(Thread::class)
  ->defaultSort('title')
  ->allowedSorts(['like_count', 'reply_count'])
  ->get();
```

In this case threads will be sorted by title because there's no sort specified in the URL.

Including relationships

```
// GET /threads?include=author
$threads = QueryBuilder::for(Thread::class)
  ->allowedIncludes(['author', 'tags'])
  ->get();
```

It's easy... And finally the spare fieldsets.

Sparse fieldsets

```
// GET /threads?fields[threads]=id,title
$threads = QueryBuilder::for(User::class)
  ->allowedFields(['id', 'title'])
  ->get();
```

As you can see, it's a really great, and powerful package. WIth this knowledge let's start our sample API using these principles.

Other ideas

API versioning

If you look inside the repository you can see that we have a file called v1.php in the routes folder. This file contains all of our API v1 routes. If we need to change our API, we can create a new v2.php and put the API v2 routes inside it. You can configure it in the RouteServiceProvider::boot() method:

```
public function boot()
{
  $this->configureRateLimiting();

  $this->routes(function () {
    Route::prefix('api/v1')
      ->middleware(['api', 'auth:sanctum'])
      ->group(base_path('routes/v1.php'));
  });
}
```

We can specify the middlewares as well, so in the v1.php we just list our routes:

```
use App\Http\Controllers\CategoryController;
use App\Http\Controllers\PopularThreadController;
use App\Http\Controllers\ReplyController;
use App\Http\Controllers\ThreadController;
use Illuminate\Support\Facades\Route;

Route::get('categories/{category}/threads/popular', [
  PopularThreadController::class,
  'index'
]);

Route::apiResource('categories', CategoryController::class)
  ->only(['index', 'store', 'destroy']);

Route::apiResource('categories/{category}/threads', ThreadController::class)
  ->except('update');

Route::apiResource(
  'categories/{category}/threads/{thread}/replies',
  ReplyController::class
)->only(['index', 'store', 'destroy']);
```

I like using nested resources and nested routes, something like:

```
GET /api/v1/categories/my-category/threads
```

This helps me to keep organized and also to write simple controllers that stick to the basic API resource methods:

Controller Method	Route
index	`GET /categories`
show	`GET /categories/my-category`
store	`POST /categories`
update	`PUT /categories/my-category`
destroy	`DELETE /categories/my-category`

API Best Practices

After all these fundamentals, let me share some of my API "best practices" with you:

- Write easy-to-understand, resourceful APIs.
- Use nested resources with nested controllers whenever it makes sense.
- Use the HTTP status codes. For example, if your endpoint is asynchronous (meaning it dispatches jobs that will take time) don't return 200 but 202 - Accepted and a link where the client can request the status of the operation.
- Don't expose auto-increment integer IDs! This is an important one. Using IDs like 1, 2 in your URLs is a security concern and can be exploited. It's especially important if your API is public. This is how I do it:
  - I have integer IDs in the database. It's mostly MySQL which is optimized for using auto-increment IDs.
  - I also have UUIDs for almost every model.
  - In the API I only expose the UUIDs.
- Versioning your API. This is also more important if you write a public API. I show you later how to do it easily with Laravel's RouteServiceProvider.
- Let the client decide what it wants. This means:
  - Loading relationships
  - FIltering
  - Sorting
  - Sparse fieldsets
- Try to standardize these things so you have the same solutions in your projects. In this case, we're gonna use JSON API.
- Make your response easy to use for the client, for example:
  - Use `camelCase` which is very natural in Javascript.
  - Use nested attributes for cohesive data.

All of the above is easy to achieve in Laravel with the help of a few packages.

To sum it up, we want API endpoints like these:

```
GET /api/v1/employees/a3b5ce95-c9c8-40f7-b8d7-06133c768a92?
include=department,paychecks

GET /api/v1/emplyoees/a3b5ce95-c9c8-40f7-b8d7-06133c768a92/paychecks?sort=-
payed_at
```

Test-Driven Development

Test-Driven development is a programming approach where the tests drive the actual implementation code. In practice, this means that we write the tests first and then the production code. I know it sounds complicated and maybe a little bit scary. So I try to rephrase it:

In Test-Driven Development the first step is to specify how you want to use your class / function. Only after that do you start writing the actual code.

The red

Let me illustrate it with a very simple example. I need to write a Calculator class that needs to be able to divide two numbers. Let's use it before we write it!

```
$calculator = new Calculator();

// I expept to be 5.00
$result = $calculator->divide(10, 2);

// I expect to be 3.33
$result = $calculator->divide(10, 3);

// I expect to be 0.00 instead of an Exception
$result = $calculator->divide(10, 0);
```

After 3 examples we have the specification:

- Always returns float with 2 decimals
- Returns 0 when division by zero happens

Instead of comments, we can use so-called assertions from PHPUnit:

```
$calculator = new Calculator();

$result = $calculator->divide(10, 2);
$this->assertEquals(5.00, $result);

$result = $calculator->divide(10, 3);
$this->assertEquals(3.33, $result);

$result = $calculator->divide(10, 0);
$this->assertEquals(0.00, $result);
```

Now, that we're using asserts we need a context. This is our test class:

```
class CalculatorTest extends TestCase
{
  public function testDivide()
  {
    $calculator = new Calculator();

    $this->assertEquals(5.00, $calculator->divide(10, 2));
    $this->assertEquals(3.33, $calculator->divide(10, 3));
    $this->assertEquals(0.00, $calculator->divide(10, 0));
  }
}
```

It's always a good idea to separate your test functions based on 'use-cases'. In this basic example we have two different scenarios:

- Divide two valid numbers
- Handle division by zero

So we should create two test functions:

```
class CalculatorTest extends TestCase
{
  public function testDivideValidNumbers()
  {
    $calculator = new Calculator();

    $this->assertEquals(5.00, $calculator->divide(10, 2));
    $this->assertEquals(3.33, $calculator->divide(10, 3));
  }

  public function testDivideWhenTheDividerIsZero()
  {
    $calculator = new Calculator();

    $this->assertEquals(0.00, $calculator->divide(10, 0));
  }
}
```

I don't know about you, but I'm going crazy just by looking at this function name: `testDivideWhenTheDividerIsZero`. Fortunately, PHPUnit has a solution for us:

```
class CalculatorTest extends TestCase
{
  /** @test */
  public function it_should_divide_valid_numbers()
  {
  }

  /** @test */
  public function it_should_return_zero_when_the_divider_is_zero()
  {
  }
}
```

With the `@test` annotation you can omit the `test` word from the function name and by using underscores you can actually read it. As you can see a test function's name reads like an English sentence: It should divide valid numbers.

A test should read like documentation for a class / method.

In this stage, you have no real code, only tests. So, naturally, these tests will fail (the Calculator class doesn't even exist).

This stage is called red because your tests fail.

The green

It's time to write some code! After we wrote the tests, we can write the actual code:

```
class Calculator
{
  public function divide(float $a, float $b): float
  {
    if ($b === 0.0) {
      return 0;
    } else {
      return round($a / $b, 2);
    }
  }
}
```

In this stage, your main focus should not be to write the most beautiful code ever. No, you want to write the minimum amount of code that makes the tests green. After your tests pass maybe you come up with a new edge-case so you should write another test, and then handle this edge-case in your code.

This stage is called green because your tests pass.

The refactor

So we wrote the minimum amount of code to make the tests pass. This means that we have an automated test for every use case we know right now. If you think about it, this is a huge benefit! From that point you are almost unable to make bugs, so you can start refactoring:

```
class Calculator
{
  public function divide(float $a, float $b): float
  {
    try {
      return round($a / $b, 2);
    } catch (DivisionByZeroError) {
      return 0;
    }
  }
}
```

This stage is called refactor because you don't write new code but make the existing codebase better.

Thank you

Thank you very much for reading this short PDF! I hope you have learned a few cool techniques.

By the way, I'm Martin Joo, a software engineer since 2012. I'm passionate about Domain-Driven Design, Test-Driven Development, Laravel, and beautiful code in general.

I released a ~100 pages eBook called **Test-Driven APIs with Laravel and Pest** where I discuss things like how you can:

- Achieve near 100% code coverage using Pest
- Write standardized requests and responses using JSON API
- Create developer-friendly APIs using nested resources
- Utilize PHP 8.1 enums with the factory and the strategy design pattern
- Use simple, yet powerful concepts from Domain-Driven Design
- ...and a lot more about API design

If you liked this PDF you should check it out. You can download a free chapter.