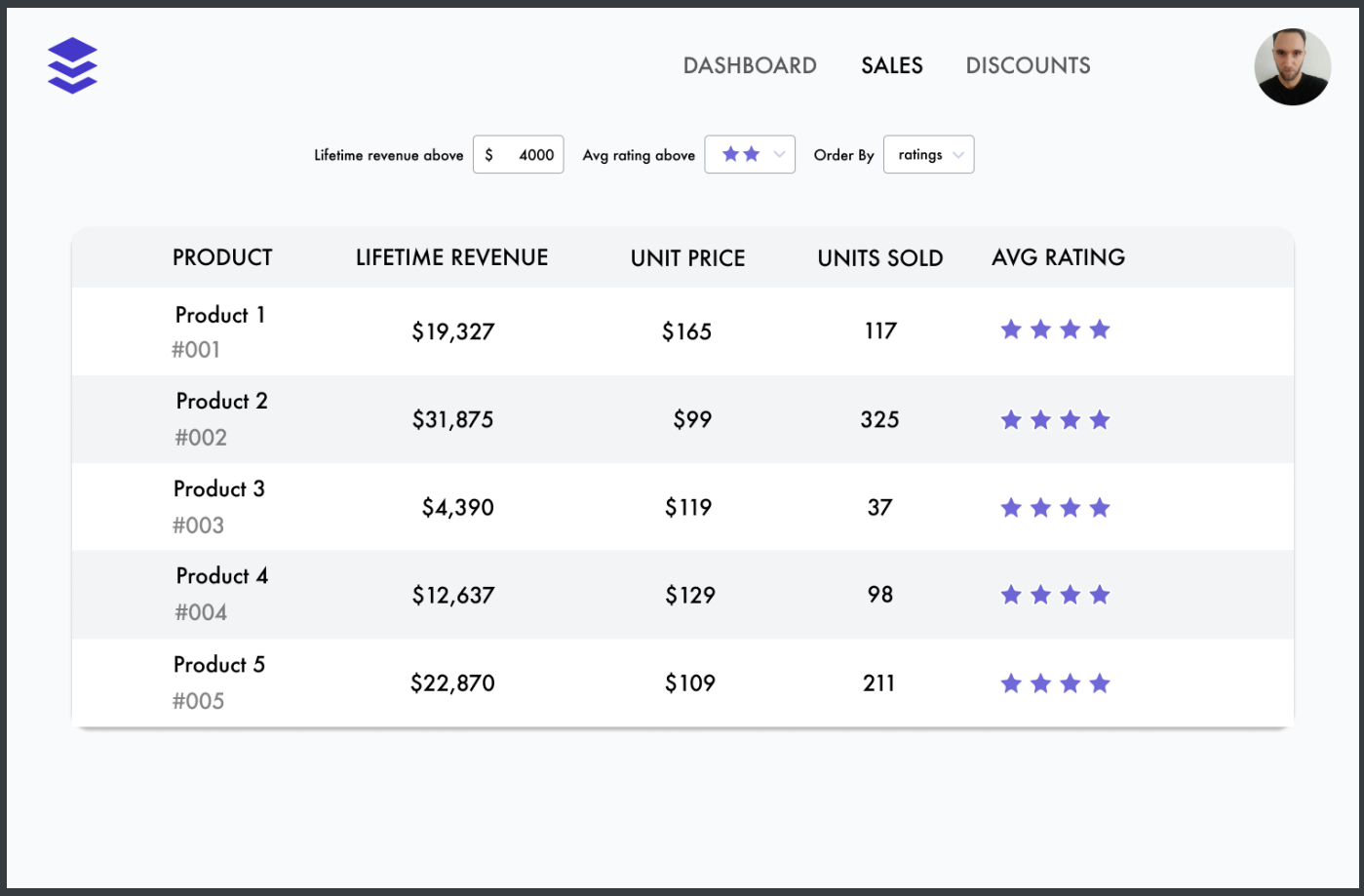
Advanced Product Filters

In this chapter I'd like to talk about applying more complicated filters and sorting products. Since it's such a common scenario, I think you'll benefit from it.

Let's say we are working on the backend of an e-commerce app, so we're building dashboards and lists for business people. Often they want more sophisticated filters than regular users. Now imagine you have 20 of those. It's easy to mess things up because these things often start as a few `where` expressions in your controller. But as the number of filters and their complexity grows your code becomes messier and you'll end up with a "don't touch it" class. You know, when you say to new developers: "This controller takes care of filtering products. It works, don't touch it!"

In this essay, I'd like to give some ideas about how to deal with these situations. Take a look at the design:



This list has two filters:

- Lifetime revenue
- Average rating

And one bonus is called the "most popular." It returns products that have:

- Higher than average revenue
- More ratings than the average
- Better than average ratings

And four kinds of sorting:

- Lifetime revenue
- Average rating
- Quantity (units sold)
- Revenue contribution

The plan is simple: **implement these features in an easy to extend and maintainable way.** The best way to achieve this is to use:

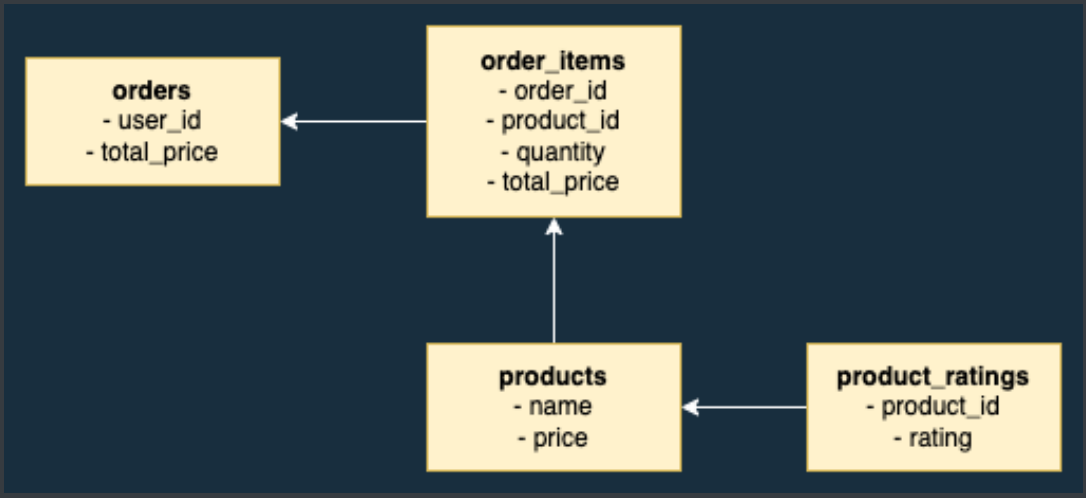
- The strategy design pattern
    - Dedicated classes for filters and sorters
- Enums
    - One for filters another for sorters. They will act as factories.
- And sometimes pipelines
    - To put everything together in a functional way.

Data modeling

The database layer is relatively straightforward:

- orders
- order_items
- products
- product_ratings
- users

That's all we need to implement these features:



Some example queries using these tables:

```sql
/* All-time revenue */
select sum(total_price)
from orders

/* Products ordered by total revenue */
select sum(total_price) as total_revenue
from products
inner join order_items
  on order_items.product_id = products.id
group by products.id
order by total_revenue desc
```

Filters

First, without any additional context or "architecture" let's just start with the Eloquent and SQL queries. After that, we're going to talk about where to put and how to structure these classes.

Revenue Filter

Users want to filter products that have a lifetime revenue above a certain threshold.

This is one way of implementing this filter using a sub-query:

```
select
  `products`.*,
  (
    select sum(`order_items`.`total_price`)
    from `order_items`
    where `products`.`id` = `order_items`.`product_id`
  ) as `total_revenue`
from `products`
group by `products`.`id`
having `total_revenue` ≥ 990
```

Another way would be to join the `order_items` table instead of using a sub-query:

```
select
  `products`.*,
  sum(order_items.total_price) as total_revenue
from `products`
inner join order_items on order_items.product_id = products.id
group by `products`.`id`
having `total_revenue` ≥ 990
```

There's not a big difference between these queries, however, there are some important things:

- Usually `join` wins when it comes to performance. However, it depends on a lot of factors such as MySQL or Postgres, exact versions, your table structure, indexes, etc.
- For me, a sub-query seems more logical. I guess it's different for everyone.
- By default, a lot of Laravel helpers use sub-queries. So if you want to use `join` you have to write custom code in some situations. I'll show you an example, in a minute.

Here's the implementation using join:

```
return Product::query()
  ->select('products.*',
DB::raw('SUM(order_items.total_price)'))
  ->join('order_items', 'products.id', '=',
'order_items.product_id')
  ->groupBy('products.id')
  ->having(DB::raw('SUM(order_items.total_price)'), '≥', 990);
```

It groups the rows by product ID and then adds a `having` clause using the sum of total prices. Since it has a `group by` we need to use `having` instead of `where`.

If you set up relationships correctly (a `Product` has many `OrderItem` in this case) you can use the `withSum` helper:

```
return Product::query()
    ->withSum('order_items as total_revenue', 'total_price')
    ->having('total_revenue', '≥', 990);
```

Laravel comes with a handful of relationship aggregate helpers such as `withSum`, `withAvg`, `withCount`, and so on. This query uses a sub-query as discussed earlier. The alias `as total_revenue` will be included in the result as a property. If you don't specify it, you can access the values as `order_items_sum_total_price`, which is created by Laravel using the following logic:

- `order_items` is the table of the relationship
- `sum` is the aggregate function you're using
- `total_price` is the column you want to sum (the second argument of the `withSum` method)

Average Rating Filter

This filter is very similar to the previous one. This is the SQL query:

```
select
  `products`.*,
  (
    select avg(`product_ratings`.`rating`)
    from `product_ratings`
    where `products`.`id` = `product_ratings`.`product_id`
  ) as `avg_rating`
from `products`
group by `products`.`id`
having `avg_rating` ≥ 4
order by `avg_rating` desc
```

And this is the Eloquent query:

```
return Product::query()
  ->withAvg('ratings as avg_rating', 'rating')
  ->having('avg_rating', '≥', 4);
```

Most Popular Filter

This filter returns products that have:

- Higher than average revenue
- More ratings than the average
- Better than average ratings

So first, we need to calculate the average values:

```
$numberOfProducts = Product::count();

$averageRevenue = Order::sum('total_price') /
$numberOfProducts;
$averageRating = ProductRating::avg('rating');
$averageNumberOfRatings = ProductRating::count() /
$numberOfProducts;
```

It's very straightforward. After we have these values we can apply them in the actual query:

```
Product::query()
    ->withSum('order_items as total_revenue', 'total_price')
    ->withAvg('ratings as avg_rating', 'rating')
    ->withCount('ratings as count_ratings')
    ->having('count_ratings', '≥', $averageNumberOfRatings)
    ->having('avg_rating', '≥', $averageRating)
    ->having('total_revenue', '≥', $averageRevenue);
```

We'll get back to filters soon, but first, let's discuss the sorters.

Sorters

Revenue Contribution Sorter

This sorter will sort products by their revenue contribution. If the total revenue is $1000 and 'Product A' made $300 in sales while Product B made $700, then the contributions are:

- Product A: 30%
- Product B: 70%

This is what the query looks like:

```
$totalRevenue = Order::sum('total_price');

Product::query()
  ->selectRaw("SUM(order_items.total_price) / $totalRevenue as
revenue_contribution")
  ->join('order_items', 'products.id', '=',
'order_items.product_id')
  ->groupBy('products.id')
  ->orderBy('revenue_contribution');
```

As you can see, the revenue contribution is calculated by this line:

```
"SUM(order_items.total_price) / $totalRevenue as
revenue_contribution"
```

It sums up the order items associated with a product and then divides this number by the total revenue. Since the `$totalRevenue` is a variable outside of the query, I choose to use a `join`, since it's the most simple way to write this query.

Average Rating and Revenue Sorter

This will sort the products based on their average ratings. This and the revenue sorter are the most simple ones:

```
// Average rating
Product::query()
  ->withAvg('ratings as avg_rating', 'rating')
  ->orderBy('avg_rating');

// Total revenue
Product::query()
  ->withSum('order_items as sum_revenue', 'total_price')
  ->orderBy('sum_revenue');
```

Strategy

If you think about it all filters are the same, and so as all sorters. They do the same thing, but with a different "strategy." For example, the revenue filter and average rating filter. They both filter out products based on some values. This means they can have the same interface but different implementations. The same is true for sorters.

Before we jump into the details, let's imagine how we want to use these classes. Let's say we follow the JSON API standard, so the request URL looks something like that:

```
/api/products?filter[revenue]=90&filter[avg_rating]=3.7
```

This request means the user wants to see every product that has:

- At least $90 in revenue
- And 3.7 or better rating

Now let's imagine a code behind this API:

```
class ProductController
{
  public function index(Request $request)
  {
    /**
     * [
     *     'revenue' => 90,
     *     'avg_rating' => 3.7,
     * ]
     */
    $filters = $request->collect('filters');

    // select * from products
```

```
        $query = Product::query();

        foreach ($filters as $name => $value) {
            // Each filter has a class such as RevenueFilter
            $filter = FilterFactory::create($name, $value);

            /**
             * Each filter class will append
             * clauses to the base query
             */
            $filter->handle($query);
        }

        return $query->get();
    }
}
```

So I'd like to see a simple for loop where we can go through the filters, create a class based on the filter's name, and then call a function. Something like `$filter->handle($query)` and each of these `handle` calls will append the appropriate `withSum`, `withAvg`, and `having` clauses to the base query.

From this little example, we know how to filter interface will look like:

- It has probably only one function, `handle`
- The constructor (called by the factory) will take a `$value`. This is the threshold value used in the `having` or `where` clause.
- `handle` takes a query and modifies it

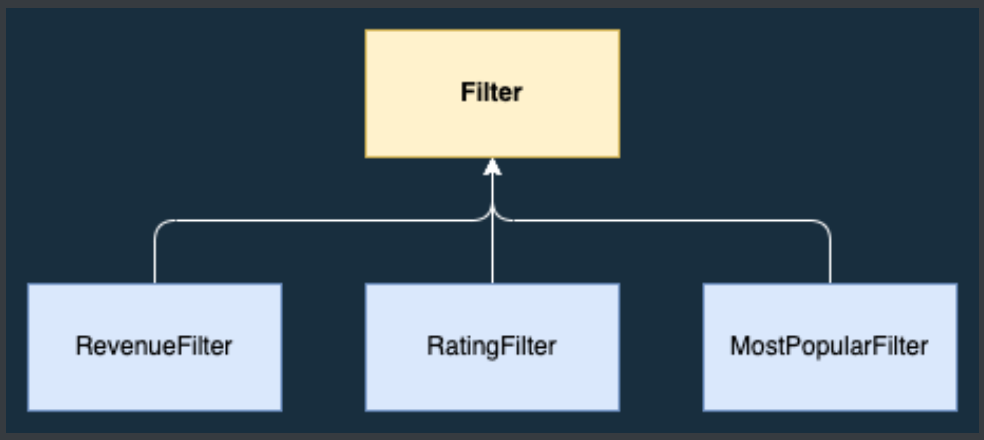
We can come up with something like this:

```php
namespace App\Filters;

use Illuminate\Contracts\Database\Eloquent\Builder;

abstract class Filter
{
  public function __construct(protected readonly int $value)
  {
  }

  abstract function handle(Builder $query): void;
}
```

And the `RevenueFilter` class looks like this:

```php
namespace App\Filters;

class RevenueFilter extends Filter
{
  function handle(Builder $query): void
  {
    $query
      ->withSum('order_items as total_revenue', 'total_price')
      ->having('total_revenue', '≥', $this->value);
  }
}
```

This is what it looks like on a diagram:



The benefits of the strategy pattern:

- Whenever you need a new implementation (a new filter in this case) you just have to add a new class. So you don't have to modify existing classes and functions. The only place you need to change is the factory (we'll talk about it in a minute). This is the letter "O" in SOLID.
- Separation of concerns. Each filter has its own class. This is a huge benefit in a large application! Just think about the `MostPopularFilter`. If it's going to change in the future (and gets much more complicated) there's a guarantee that you won't cause bugs in other filters. They are completely separated.
- Single responsibility. Each filter class does only one thing: filter by X criteria. They don't know anything about the outside world, they just do their well-defined job. This is the letter "S" in SOLID.
- Easy interaction. The controller in the above example only interacts with the interface of the `Filter` abstract class. It doesn't matter if the user wants to order by revenue or ratings. It doesn't matter. The interaction is always the same. This is the letter "D" in SOLID.
- Small interface. Since filter classes are so well-defined they only require one method and a constructor to work. It's easy to use, and easy to maintain. This is the letter "I" from SOLID.

From a practical point of view the biggest benefit is that if you need to handle a new kind of filter, you just create a new class from scratch, and add a new line to a factory.

Enums

I talked about factories on the previous pages but in fact, in PHP 8.1 we can use enums to achieve the same. For example, this is the `Filters` enum:

```
namespace App\Enums;

enum Filters: string
{
  case Revenue = 'revenue';
  case AverageRating = 'avg_rating';
  case MostPopular = 'most_popular';

  public function createFilter(int $value): Filter
  {
    return match ($this) {
      self::Revenue => new RevenueFilter($value),
      self::AverageRating => new AverageRatingFilter($value),
      self::MostPopular => new MostPopularFilter($value),
    };
  }
}
```

Fortunately, enums can contain methods, and the `createFilter` will behave just like a factory. It can be used such as:

```
$filter = Filters::from('revenue')->createFilter(99);
$filter->handle();
```

As you can imagine, the values 'revenue' and 99 will come from the request and will be handled by the controller.

We can do the same thing with Sorters as well:

```
namespace App\Enums;

enum Sorters: string
{
  case Rating = 'rating';
  case Revenue = 'revenue';
  case Quantity = 'quantity';
  case RevenueContribution = 'revenue_contribution';

  public function createSorter(SortDirections $sortDirection):
Sorter
  {
    return match ($this) {
      self::Rating => new RatingSorter($sortDirection),
      self::Revenue => new RevenueSorter($sortDirection),
      self::Quantity => new QuantitySorter($sortDirection),
      self::RevenueContribution => new
RevenueContributionSorter($sortDirection),
    };
  }
}
```

It follows the same logic. The only difference is that a sorter doesn't need a value. It only needs a direction (asc or desc). This is what the `SortDirection` is:

```php
namespace App\Enums;

enum SortDirections: string
{
  case Desc = 'desc';
  case Asc = 'asc';
}
```

By the way, I didn't list it here but the `Sorter` class and the subclasses follow exactly the same structure as the `Filter` classes. They have only one method that takes a query and modifies it (adds the order by clause to it).

Pipelines

Right now, with the `Filter` `Sorter` class and enums we can write something like this:

```
class ProductController
{
  public function index(Request $request)
  {
    $filters = $request->collect('filters');

    $query = Product::query();

    foreach ($filters as $name => $value) {
      $filter = Filters::from($name)->createFilter($value);

      $filter->handle($query);
    }

    return $query->get();
  }
}
```

It's very clean. However, we can use Laravel pipelines to make it even more "generic:"

```
use Illuminate\Pipeline\Pipeline;

class ProductController
{
  public function index(Request $request)
  {
    $filters = $request->collect('filters')
      ->map(fn (int $value, string $name) =>
        Filters::from($name)->createFilter($value)
      )
      ->values();

    return app(Pipeline::class)
      ->send(Product::select('products.*'))
      ->through($filters)
      ->thenReturn()
      ->get();
  }
}
```

A pipeline has multiple "stops" or pipes. These pipes are classes that do something with the initial value. In this example, these pipes are the filter classes, and the initial value is a product query builder object. You can of the whole flow like this:

- We have an initial query builder instance.
- We send this object through the pipes. Or in other words, through a collection of `Filter` instances.
- Every `Filter` modifies the initial query and returns a new one (we need to make a slight change in our classes).
- After each `Filter` has finished its job the `thenReturn` will return the final query builder instance.

- After we have the builder, the `get` will run the actual query and returns a collection.

Why is it better than a `foreach`?

First of all, it's not better. It's just a different approach. However, I think it has one advantage. A `foreach` is very "hackable." What I mean by this is that it encourages developers to write if-else statements, nested loops, break statements, and other "stuff" to handle edge cases, or apply a "quick fix" here and there. By using a pipeline you cannot do any of those! The whole flow is "closed." So if you need a "quick fix" because your new Filter class is not compatible with the current architecture you have to think about why is it the case, and how you can solve it. You need to make it compatible with the current solution, or you need to restructure the existing filters and drop the pipeline approach. So I think using a Pipeline helps us follow the Open-Closed Principle from SOLID. It's open for new filters but closed for "quick fixes."

To use the `Filter` classes in a pipeline, we need to make a small change:

```
namespace App\Filters;

use Closure;

abstract class Filter
{
  public function __construct(protected readonly int $value)
  {
  }

  abstract function handle(Builder $query, Closure $next):
Builder;
}
```

The handle method now takes a second argument called `$next` and returns a `Builder` instance. The `$next` is very similar to a middleware. Each middleware calls the next one via a closure. The same applies here as well. Each pipe triggers the next one by invoking the `$next` closure:

```
namespace App\Filters;

use Closure;

class RevenueFilter extends Filter
{
  function handle(Builder $query, Closure $next): Builder
  {
    $query
      ->withSum('order_items as total_revenue', 'total_price')
      ->having('total_revenue', '≥', $this->value);

    return $next($query);
  }
}
```

And it also needs to return the result of `$next`. The same logic applies to `Sorter` classes. Now that we have everything ready it's time to implement the controller and an action.

Action & Controller

As always I want to keep my controllers as small as possible. They have only three responsibilities, in my opinion:

- Accepting the request
- Invoking other classes
- Returning a response

This is an example of a request URL:

```
api/products
  ?filter[revenue]=90
  &filter[avg_rating]=3.7
  &sort=revenue
  &sort_direction=asc
```

Which means:

- Products with more than $90 revenue
- Products with more than a 3.7 average rating
- Sorted by revenue in ascending order

This is what the `ProductController` looks like:

```
namespace App\Http\Controllers;

use App\Actions\FilterProductsAction;
use App\Http\Requests\GetProductsRequest;

class ProductController extends Controller
{
```

```
  public function index(GetProductsRequest $request)
  {
    return FilterProductsAction::execute(
      $request->collect('filter'),
      $request->sorter(),
      $request->sortDirection(),
    );
  }
}
```

Those getters in the request convert strings to enums and apply some default values:

```
class GetProductsRequest extends FormRequest
{
  public function sortDirection(): SortDirections
  {
    if (!$this->sort_direction) {
      return SortDirections::Desc;
    }

    return SortDirections::from($this->sort_direction);
  }

  public function sorter(): Sorters
  {
    if (!$this->sort) {
      return Sorters::Rating;
    }

    return Sorters::from($this->sort);
```

```
  }
}
```

The last piece of the puzzle is the `FilterProductsAction`. You have already seen this class without knowing it. This is the one where construct the pipeline:

```
namespace App\Actions;

class FilterProductsAction
{
  /**
   * @param Collection<string, int> $filterValues
   * @return Collection<Product>
   */
  public static function execute(
    Collection $filterValues,
    Sorters $sort,
    SortDirections $sortDirection
  ): Collection {

    $filters = $filterValues
      ->map(fn (int $value, string $name) =>
        Filters::from($name)->createFilter($value)
      )
      ->values();

    return app(Pipeline::class)
      ->send(Product::select('products.*'))
      ->through([
        ...$filters,
```

```
            $sort->createSorter($sortDirection)
        ])
        ->thenReturn()
        ->get();
    }
}
```

`$filterValues` is a Collection like this:

```
[
    'revenue' => 90,
    'avg_rating' => 3.7,
];
```

So it contains the filter names and the values associated with them. In the pipeline setting there's only one new thing, this line:

```
->through([ ... $filters, $sort->createSorter($sortDirection)])
```

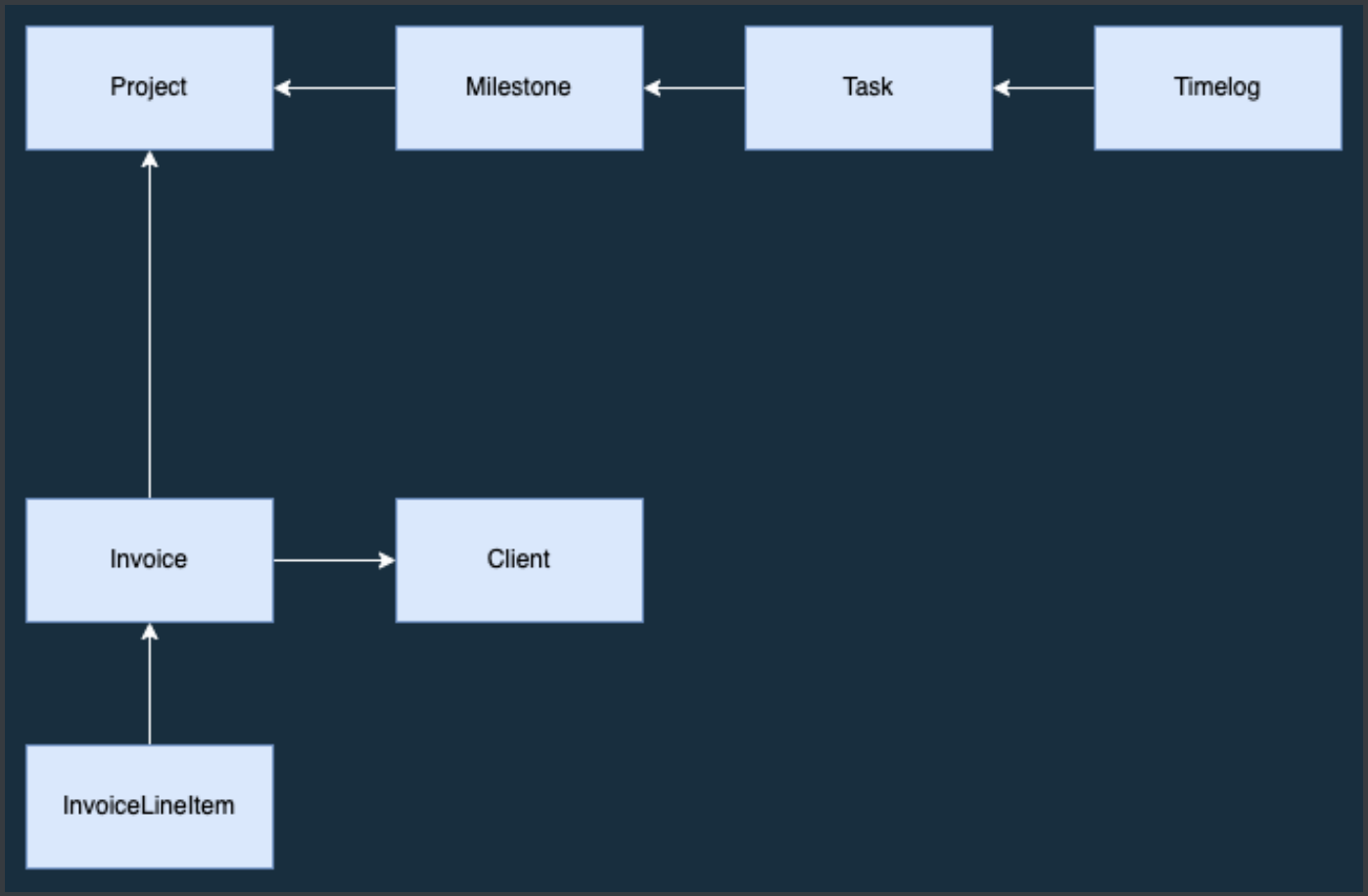
Here we want to send the query through not only the filter but also the sorter. This is why I create a new array that contains both.

Boundaries in DDD and Modular Systems

This chapter has a back story. In April 2022 I released a book called Domain-Driven Design with Laravel. DDD has some technical and strategic aspects. Technical means that we have some classes and concepts we can use in our code to write better software. Strategic means that DDD tries to close the gap between the business language and the code. But DDD also cares about boundaries. It's a crucial attribute of domain-driven design and I left it out of the book. It was intentional. I took a risk, to be honest because a lot of people think if you're not obeying these boundaries it's not real DDD. I got a few complaints about it, and three or four refund requests. I refunded everyone and I completely respect their perspectives.

So what are these boundaries and why did I leave them out of the book? In domain-driven design we have domains. A domain is something like a module. It groups classes together. But only classes that are related to each other.

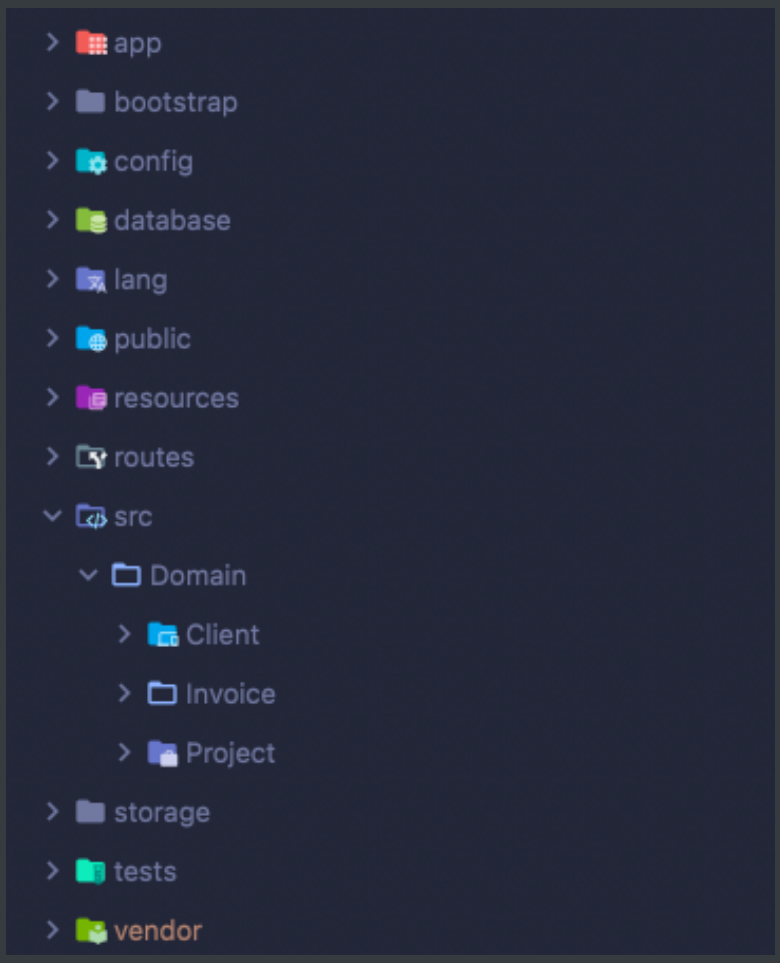
Let's see an example. We are working on a project management application where we can track our time and bill a customer. So the app has models like these:



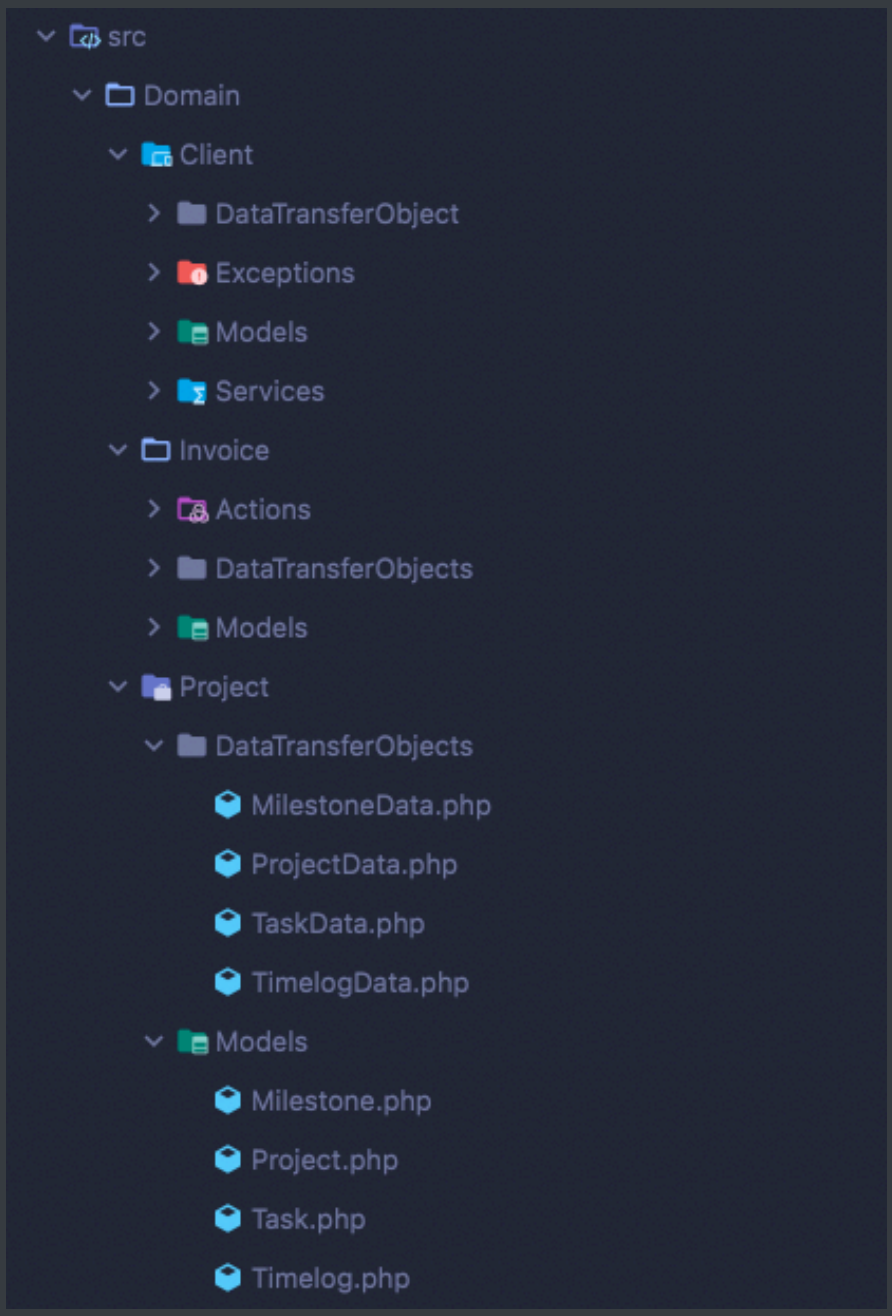
These classes can be grouped by their "type":

Class	Group
Project	Project
Milestone	Project
Task	Project
Timelog	Project
Invoice	Invoice
InvoiceLineItem	Invoice
Client	Client
Other (not so important) client-related models	Client

These groups are the domains in the application. The models and other classes inside the 'Project' domain can be grouped together and isolated from the other domains. These classes are responsible for handling project management-related user stories. Nothing else matters to them (Metallica shout-out).



And each domain contains **only business logic related classes**. This means no controllers, commands, migrations, or anything specific to the application or infrastructure. For example, a controller is specific to a web or API application, meanwhile, a command is specific to a console application. They can only exist in the context of those applications. On the other hand, the `Project` model **does not care about** if it's being used by a command, API controller, web controller, Inertia controller, migration, SPA, or MVC app. It's context-independent by nature. So it lives inside the domain folder:



These are domains in a nutshell. However, in this article, I'd like to focus on boundaries specifically. If this is the first time you hear about domains and applications you can read this article which describes them in great detail.

Life Without Boundaries

Let's stick to the project management example and implement a simple action that creates an invoice for an entire milestone. The relationship between the models are:

- `Project` -> has many -> `Milestone` -> has many -> `Task` -> has many -> `Timelog`
- `Project` -> has many `Invoice`
- `Invoice` -> belongs to one -> `Project`
- `Invoice` -> has many -> `InvoiceLineItem`
- `Project` -> belongs to one -> `Client`

So the flow looks like this:

- Create an invoice for the client
- Get the tasks associated with the milestone
- Create an invoice line item for each task (based on the time logs for that task)
- Calculate the total amount for the invoice

```
namespace Domain\Invoice\Actions;

class CreateMilestoneInvoiceAction
{
  public function execute(Milestone $milestone): Invoice
  {
    $invoice = Invoice::create([
      'client_id' => $milestone->project->client_id,
      'client_name' => $milestone->project->client->full_name,
      'project_id' => $milestone->project_id,
    ]);

    $invoiceAmount = 0;

    foreach ($milestone->tasks as $task) {
```

```
      $lineItem = $invoice->addLineItem($task);

      $invoiceAmount += $lineItem->total_amount;
    }

    $invoice->total_amount = $invoiceAmount;
    $invoice->save();

    return $invoice;
  }
}

class Invoice extends Model
{
  public function addLineItem(Task $task): InvoiceLineItem
  {
    $hoursLogged = $task->timelogs->sum('hours');

    return InvoiceLineItem::create([
      'invoice_id' => $this->id,
      'task_id' => $task->id,
      'item_name' => $task->name,
      'item_quantity' => $hoursLogged,
      'total_amount' => $hoursLogged * 30,
    ]);
  }
}
```

This is a very straightforward class. This is the usual Laravel code you're probably used to. It violates boundaries at least four times:

- Violation #1: This class is inside the `Invoice` domain, so it **should not** access the `Milestone` model directly.
- Violation #2: It **should not** use the `project` relationship from the `Milestone` model, and the `client` relationship from the `Project` model. We are exposing too much information to the `Invoice` domain. It **should not** know anything about the inner structures of models from another domain.
- Violation #3: The `Invoice` model **should not** access the `Task` model and its `timelogs` relationship. Same problem as Violation #2.
- Violation #4: Under the hood, the `client_id` and `project_id` columns in the `invoices` table are foreign keys to the `clients` and `projects` table. A table in the invoice domain **should not** reference another table from the client or project domain.

So if you're writing code like this, you will never enter the gates of DDD Walhalla. Now let's make this code DDD-compatible.

Life With Boundaries

Violation #1

First, let's get rid of the `Milestone` argument. We cannot pass models across domains. Fortunately, we can express every model as a DTO or data transfer object.

The first step is to create a `MilestoneData` class:

```
namespace Domain\Project\DataTransferObjects;

use Spatie\LaravelData\Data;

class MilestoneData extends Data
{
  public function __construct(
    public readonly ?int $id,
    public readonly string $name,
    public readonly Lazy|ProjectData $project,
    /** @var DataCollection<TaskData> */
    public readonly Lazy|DataCollection $tasks,
  ) {}

  public static function fromModel(Milestone $milestone): self
  {
    return self::from([
      ...$milestone->toArray(),
      'project' => Lazy::whenLoaded(
        'project',
        $milestone,
        fn () => ProjectData::from($milestone->project)
```

```
      ),
      'tasks' => Lazy::whenLoaded(
        'tasks',
        $milestone,
        fn () => TaskData::collection($milestone->tasks)
      ),
    ]);
  }
}
```

In this example, I use the `laravel-data` package by Spatie. If you're confused about these `Lazy` things, please check out the documentation, but it's not important for this article. In a nutshell, `Lazy::whenLoaded` is very similar to Laravel's Resource `whenLoaded` function. So in the example above, the `project` will only be included if the relationship is already eager-loaded in the `$milestone` instance. It helps us to avoid N+1 query problems.

Now that we have the `MilestoneData` we can use it in the action:

```
namespace Domain\Invoice\Actions;

use Domain\Project\DataTransferObjects\MilestoneData;

class CreateMilestoneInvoiceAction
{
  public function execute(MilestoneData $milestone)
  {
    // ...
  }
}
```

So after this small refactor we're no longer exposing too much information from the project domain. DTOs are meant to be transferring data between components. They don't have functions like:

- delete
- update
- relationships

They don't contain every column and behavior such as a Model. So it's a lot safer to use them, and it's harder to write fragile applications.

Violation #2

Earlier we used the milestone's project relationship to access client information. Instead of accessing the client directly through relationships, we have to introduce a `ClientService` that takes an integer ID and returns a `ClientData` DTO:

```
namespace Domain\Client\Services;

class ClientService
{
  /**
   * @throws ClientNotFoundException
   */
  public function getClientById(int $clientId): ClientData
  {
    $client = Client::find($clientId);

    if (!$client) {
      throw new ClientNotFoundException(
        "Client not find with id: $clientId"
      );
    }

    return ClientData::from($client);
  }
}
```

With this class we can eliminate the relationships from the action:

```
namespace Domain\Invoice\Actions;

class CreateMilestoneInvoiceAction
{
  public function __construct(
    private readonly ClientService $clientService
  ) {}

  public function execute(
    MilestoneData $milestone
  ): InvoiceData {
    $client = $this->clientService->getClientById(
      $milestone->project->resolve()->client_id
    );

    $invoice = Invoice::create([
      'client_id' => $client->id,
      'client_name' => $client->full_name,
      'project_id' => $milestone->project->resolve()->id,
    ]);

    // ...
  }
}
```

Now instead of accessing relationships and exposing a `Client` instance we only use DTOs and an integer ID. The `resolve` function is `laravel-data` specific. It resolves the value from a `Lazy` instance, so it just returns a `ProjectData` DTO.

Violation #3

The next violation was in the `Invoice` model. The `addLineItem` method accessed the `Task` model and the `Timelog` model through a relationship. I think you can already guess the solution: use DTOs.

```
namespace Domain\Invoice\Models;

class Invoice extends Model
{
  public function addLineItem(TaskData $task): InvoiceLineItem
  {
    $hoursLogged = $task
      ->timelogs
      ->resolve()
      ->toCollection()
      ->sum('hours');

    return InvoiceLineItem::create([
      'invoice_id' => $this->id,
      'task_id' => $task->id,
      'item_name' => $task->name,
      'item_quantity' => $hoursLogged,
      'total_amount' => $hoursLogged * 30,
    ]);
  }
}
```

So I changed the `Task` model to a `TaskData` DTO. The `resolve` and `toCollection` methods come from the `laravel-data` package. It returns the `timelogs` as a Laravel collection.

I won't list the solution to violation #3 but you can in the repository that I removed the foreign keys that cross the boundaries. The resulting action looks like this:

```
namespace Domain\Invoice\Actions;

class CreateMilestoneInvoiceAction
{
  public function __construct(
    private readonly ClientService $clientService
  ) {}

  public function execute(
    MilestoneData $milestone
  ): InvoiceData {
    $client = $this->clientService->getClientById(
      $milestone->project->resolve()->client_id
    );

    $invoice = Invoice::create([
      'client_id' => $client->id,
      'client_name' => $client->full_name,
      'project_id' => $milestone->project->resolve()->id,
    ]);

    $invoiceAmount = 0;

    foreach ($milestone->tasks->resolve() as $task) {
      $lineItem = $invoice->addLineItem($task);

      $invoiceAmount += $lineItem->total_amount;
```

```
        }

        $invoice->total_amount = $invoiceAmount;
        $invoice->save();

        return InvoiceData::from($invoice->load('line_items'));
    }
}
```

Please notice that the action returns an `InvoiceData` instead of an `Invoice` model. This is a general rule you should follow if you want to respect boundaries: most actions, services, and repositories should return DTOs instead of models.

Conclusion

Why did I leave this technique out of the book? Because I truly believe it's overkill for 90% of Laravel projects. Just think about it:

- You cannot use relationships in some situations. In this example, `laravel-data` did a good job, but in the long run, you'll miss out on a lot of Eloquent features. For example, you cannot replace `withAvg` with a DTO. You have two options:
  - Using the collection to calculate an average
  - Performing an extra query by calling a service function
- It requires more database queries. For example, we had to query a `Client` from the database that was already available! By using these `getById` methods and avoiding Eloquent relationships you'll end up with N+1 query problems very, very soon. I mean, it's already a big problem in a lot of applications.
- It requires more code. Mainly because of classes like the `ClientService`.
- No foreign keys. I mean, you still can use them, but not if they violate your boundaries. The lack of foreign keys and the fact that you'll have more N+1 problems will result in very poor performance.

But here's the most important thing and the real reason why I left this out from the book. **It's easy to ruin a project with this approach**. Using this approach is not natural in Laravel or PHP. Try to implement this with six other developers where three of them are juniors. It's almost like a guarantee for failure.

And it even gets worse. This was a simplified example of boundaries. In real DDD there are three different concepts:

- Domain
- Subdomain
- Bounded context

We only used domains, so it gets even harder to identify boundaries and abstractions. Even DDD gods are arguing about these concepts. Just search for "bounded context" or "bounded context vs domain" and you'll see that every Java/C# DDD developer has a slightly different definition in their head.

In my opinion, if you're using the technical aspects of DDD and you pay attention to the strategic design you'll do fine in a larger project. So instead of boundaries, I try to focus on smaller concepts like:

- DTOs
- Value Objects
- Services
- Actions
- Domains
- Applications

And generally speaking, I'm trying to write code that reflects the business language and domain. However, if you're working on **Shopify-scale** applications you almost definitely need boundaries and those more advanced concepts. Just to be clear, when I say Shopify-scale I'm not referring to the millions of users they have. I'm talking about the **2.8 million lines of code and 500,000 commits in one monolith!** These numbers come from their engineering blog.

If you want to learn more about boundaries and monoliths, please check out this video from Laracon 2022, presented by Ryuta Hamasaki. It's a great talk!

Value Objects Everywhere

In this chapter, I'd like to talk about value objects. If you don't know what are they, here's a quick introduction.

Value Object is an elementary class that contains mainly (but not only) scalar data. So it's a wrapper class that holds together related information. Here's an example:

```
class Percent
{
  public readonly ?float $value;
  public readonly string $formatted;

  public function __construct(float $value)
  {
    $this->value = $value;

    if ($value === null) {
      $this->formatted = '';
    } else {
      $this->formatted = number_format(
        $value * 100, 2
      ) . '%';
    }
  }

  public static function from(?float $value): self
  {
    return new self($value);
  }
```

```
}
```

This class represents a percentage value. This simple class gives you three advantages:

- It encapsulates the logic that handles null values and represents them as percentages.
- You always have two decimal places (by default) in your percentages.
- Better types.

An important note: business logic or calculation is not part of a value object. The only exception I make is basic formatting.

That's it. This is a value object. It's an object that contains some values. The original definition of a value object states two more things:

- It's immutable. You have no setters and only read-only properties.
- It does not contain an ID or any other property related to the identification. Two value objects are equal only when the values are the same. This is the main difference between a VO and a DTO.

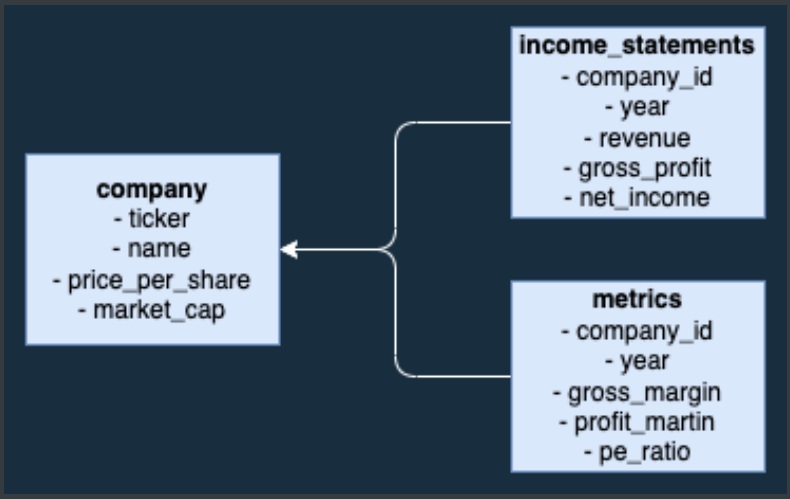
Data Modeling

To really understand value objects, we'll implement a very basic financial app. Something like Seekingalpha, Morningstar, Atom Finance, or Hypercharts. If you don't know these apps, here's a simplified introduction:

- In the app we store companies. Publicly-traded companies, such as Apple or Microsoft.
- We also store financial data, such as income statements.
- The app will calculate some important metrics from these data. For example, profit margin, gross margin, and a few others.

In the sample application, I'll only implement a handful of metrics, and I'll only store the income statements (no balance sheets or cash flows). This is more than enough to illustrate to use of value objects.

This is what the database looks like:



As you can see, it's quite easy. This is a sample row from the `companies` table:

id	ticker	name	price_per_share	market_cap
1	AAPL	Apple Inc.	14964	2420000
2	MSFT	Microsoft Inc.	27324	2040000

`price_per_share` is the current share price of the company's stock. It's stored in cent value, so `14964` is `$149.64`. This is a common practice in order to avoid rounding mistakes.

`market_cap` is the current market capitalization of the company (price_per_share * number of shares). It is stored in millions, so `2420000` is `$2,420,000,000,000` or `$2,420B` or `$2.42T`. Storing huge financial numbers in millions (or thousands in some cases) is also a common practice in financial applications.

Now let's see the `income_statements` table:

company_id	year	revenue	gross_profit
1	2022	386017	167231
1	2021	246807	167231

Each item on the income statement has its own column such as revenue or gross_profit. One row in this table describes a year for a given company. And as you can probably guess, these numbers are also in millions. So `386017` means `$386,017,000,000` or `$386B` for short.

If you're wondering why to store these numbers in millions, the answer is pretty simple: it's easier to read. Just check out Apple's page on Seekingalpha, for example:

	Sep 2012	Sep 2013	Sep 2014	Sep 2015	Sep 2016	Sep 2017	Sep 2018	Sep 2019	Sep 2020	Sep 2021	TTM
Revenues											
Revenues						229,234.0	265,595.0	260,174.0	274,515.0	365,817.0	386,017.0
Other Revenues						-	-	-	-	-	-
Total Revenues						**229,234.0**	**265,595.0**	**260,174.0**	**274,515.0**	**365,817.0**	**386,017.0**
Cost Of Revenues						141,048.0	163,756.0	161,782.0	169,559.0	212,981.0	218,786.0
Gross Profit						**88,186.0**	**101,839.0**	**98,392.0**	**104,956.0**	**152,836.0**	**167,231.0**

The `metrics` table is very similar to `income_statements`:

company_id	year	gross_margin	profit_margin	pe_ratio
1	2022	0.43	0.26	2432
2	2022	0.68	0.34	2851

Each metric has its own column, and each row represents a year for a given company. Most metrics are percentage values stored as decimals. The `pe_ratio` stands for "price/earnings ratio." If a company's share trades at $260 and its earnings are $20 per share, then the P/E ratio is 13.00. It's a decimal number stored as an integer.

Maybe you're asking "why not call it price_per_earnings_ratio?" It's a good question! In my opinion, our goal as software developers should be to write code that is as close to the business language as possible. But in the financial sector, nobody calls it "price per earnings ratio." It's just the "PE ratio." So, in fact, this is the correct language, in my opinion.

API

We want to implement three APIs.

GET /companies/{company}

It'll return the basic company profile:

```
{
  "data": {
    "id": 1,
    "ticker": "AAPL",
    "name": "Apple Inc.",
    "price_per_share": {
      "cent": 14964,
      "dollar": 149.64,
      "formatted": "$149.64"
    },
    "market_cap": {
      "millions": 2420000,
      "formatted": "2.42T"
    }
  }
}
```

It'll also return the price and market cap data in human-readable formats.

GET /companies/{company}/income-statements

It returns the income statements grouped by items and years:

```
{
  "data": {
    "years": [
      2022,
      2021
    ],
    "revenue": {
      "2022": {
        "value": 386017000000,
        "millions": 386017,
        "formatted": "386,017"
      },
      "2021": {
        "value": 246807000000,
        "millions": 246807,
        "formatted": "246,807"
      }
    },
    "eps": {
      "2022": {
        "cent": 620,
        "dollar": 6.2,
        "formatted": "$6.20"
      },
      "2021": {
        "cent": 620,
        "dollar": 6.2,
        "formatted": "$6.20"
      }
    }
```

```
    }
}
```

The right data structure will heavily depend on the exact use case and UI. This structure is pretty good for a layout similar to Seekingalpha's (the screenshot from earlier). This API also formats the values.

GET /companies/{company}/metrics

This is the API that returns the metrics:

```
{
  "data": {
    "years": [
      2022
    ],
    "gross_margin": {
      "2022": {
        "value": 0.43,
        "formatted": "43.00%",
        "top_line": {
          "value": 386017000000,
          "millions": 386017,
          "formatted": "386,017"
        },
        "bottom_line": {
          "value": 167231000000,
          "millions": 167231,
          "formatted": "167,231"
        }
      }
```

```
    },
    "pe_ratio": {
      "2022": {
        "value": "24.32"
      }
    }
  }
}
```

Each margin contains the top and bottom line information as well. In the case of gross margin, the top line is the revenue and the bottom line is the gross profit.

Identifying Value Objects

Now that we've seen the database and the API, it's time to define the value objects. If you take a closer look at the JSON you can identify five different kinds of values:

- **Ratio**. It's a simple number expressed as a float. Right now, the PE ratio is the only ratio-type data in the app.
- **Margin**. It has a raw value, a percentage, a top line, and a bottom-line value. Gross margin, operating margin, and profit_margin will use this data type.
- **Price**. It has a cent, dollar, and formatted value. Both `price_per_share` and `eps` (which is earnings per share) use this data type.
- **Market Cap**. It's a unique one because it has three different formats: `2.42T`, `242B`, and `577M`. All of these are valid numbers to express a company's market capitalization. When a company hits the trillion mark we don't want to use `1000B` but rather `1T`. SO we need to handle these cases.
- **Millions**. Every item in the income statement is expressed as millions so it makes sense to use a value object called `Millions`.

Now, take a look at these value object names! We're working on a financial app, and we'll have classes like `Millions`, `Margin`, or `MarketCap`.

This is the kind of codebase that makes sense. Even after five years.

Implementing Value Objects

Price

Price seems the most obvious so let's start with that one. The class itself is pretty straightforward:

```
class Price
{
  public readonly int $cent;
  public readonly float $dollar;
  public readonly string $formatted;

  public function __construct(int $cent)
  {
    $this->cent = $cent;

    $this->dollar = $cent / 100;

    $this->formatted = '$' . number_format($this->dollar, 2);
  }

  public static function from(int $cent): self
  {
    return new self($cent);
  }
}
```

Several important things:

- Every value object has `public readonly` properties. `readonly` makes sure they are immutable, while `public` makes them easy to access, so we don't need to write getters or setters.
- A lot of value object has a `from` factory function. It fits the overall style of Laravel very well.

This object can be used like this:

```
$company = Company::first();

$price = Price::from($company->price_per_share);
```

The next question is: how do we use this object? There are two paths we can take:

- Casting the values on the Model's level.
- Or casting them on the API's level.

Casting in models

We have at least two possible solutions to cast attributes to value objects in the models.

Using attribute accessors:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Casts\Attribute;

class Company extends Model
{
  public function pricePerShare(): Attribute
  {
    return Attribute::make(
      get: fn (int $value) => Price::from($value)
    );
  }
}
```

It's an excellent solution and can work 95% of the time. However, right we are in the remaining 5% because we have 10+ attributes we want to cast. In the `IncomeStatement` model we need to cast almost every attribute to a `Millions` instance. Just imagine how the class would look like with attribute accessors:

```
namespace App\Models;

class IncomeStatement extends Model
{
  public function pricePerShare(): Attribute
  {
    return Attribute::make(
      get: fn (int $value) => Millions::from($value)
    );
  }
```

```
  /* same code here */
  public function costOfRevenue(): Attribute {}

  /* same code here */
  public function grossProfit(): Attribute {}

  /* same code here */
  public function operatingExpenses(): Attribute {}

  // 8 more methods here
}
```

So in our case, using attribute accessors is not optimal. Fortunately, Laravel has a solution for us! We can extract the casting logic into a separate `Cast` class:

```
namespace App\Models\Casts;

use App\ValueObjects\Price;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class PriceCast implements CastsAttributes
{
  public function get($model, $key, $value, $attributes)
  {
    return Price::from($value);
  }

  public function set($model, $key, $value, $attributes)
  {
    return $value;
```

```
    }
}
```

This class does the same thing as the attribute accessor:

- `get` is called when you access a property from the model and it transforms the integer into a `Price` object.
- `set` is called when you set a property in the model before you save it. It should transform a `Price` object into an integer. But as you can see, I just left it as is because we don't need this for the example. If you return `$value` from the set method, Laravel won't do any extra work. So there's no attribute mutation.

The last step is to actually use this `Cast` inside the `Company` model:

```
class Company extends Model
{
    use HasFactory;

    protected $guarded = [];

    protected $casts = [
        'price_per_share' => PriceCast::class,
    ];
}
```

Now we can use it like this:

```
$company = Company::first();

// This is when the PriceCast::get() will be executed
$pricePerShare = $company->price_per_share;

// $127.89
echo $pricePerShare->formatted;

// 127.89
echo $pricePerShare->dollar;

// 12789
echo $pricePerShare->cent;
```

Where are we going to use them? In resources, for example:

```
namespace App\Http\Resources;

class CompanyResource extends JsonResource
{
  public function toArray($request)
  {
    return [
      'id' => $this->id,
      'ticker' => $this->ticker,
      'name' => $this->name,
      'price_per_share' => $this->price_per_share,
      'market_cap' => $this->market_cap,
    ];
```

```
    }
}
```

Since these value objects contain only public properties Laravel will automatically transform them into arrays when converting the response into JSON. So this resource will result in the following JSON response:

```
{
  "data": {
    "id": 1,
    "ticker": "AAPL",
    "name": "Apple Inc.",
    "price_per_share": {
      "cent": 14964,
      "dollar": 149.64,
      "formatted": "$149.64"
    },
    "market_cap": {
      "millions": 2420000,
      "formatted": "2.42T"
    }
  }
}
```

This is how we can cast values in Eloquent models. But we can skip this setup and cast the values directly inside resources.

Casting in resources

This is much more simple than the previous one. All we need to do is create a `Price` object inside the resource:

```
namespace App\Http\Resources;

class CompanyResource extends JsonResource
{
  public function toArray($request)
  {
    return [
      'id' => $this->id,
      'ticker' => $this->ticker,
      'name' => $this->name,
      'price_per_share' => Price::from($this->price_per_share),
      'market_cap' => MarketCap::from($this->market_cap),
    ];
  }
}
```

Now the `Company` model does not have any casts, so we just instantiate a `Price` and a `MarketCap` object from the integer values.

How to choose between the two?

- To be honest, it's hard to tell without a concrete use case.
- However, if you only need these values in the API, then maybe you can skip the whole `Cast` thing and just create a value object in resources.
- But if you need these values to handle other use-cases as well it's more convenient to use Eloquent casts. Some examples:

    - Notifications. For example, a new income statement just came out, and you want to notify your users and include some key values in the e-mail. Another example can be a price notification.
    - Queue jobs. For example, you need to recalculate price-dependent metrics and values on a scheduled basis.
    - Broadcasting via websocket. For example, the price is updated in real-time on the FE.
    - Each of these scenarios can benefit from using Eloquent `Cast` because otherwise you end instantiating these value objects in every place.
- In general, I think it's a good idea to use these objects in models. It makes your codebase more high-level, and easier to maintain.

So I'm going to use Eloquent `Cast` to handle the casting.

MarketCap

As discussed earlier, the market cap is a bit more unique, so it has its own value object. We need this data structure:

```
"market_cap": {
  "millions": 2420000,
  "formatted": "2.42T"
}
```

The `formatted` property will change based on the market cap of the company, for example:

```
"market_cap": {
  "millions": 204100,
  "formatted": "204.1B"
}
```

And the last case:

```
"market_cap": {
  "millions": 172,
  "formatted": "172M"
}
```

This is what the class looks like:

```
namespace App\ValueObjects;
```

```php
class MarketCap
{
  public readonly int $millions;
  public readonly string $formatted;

  public function __construct(int $millions)
  {
    $this->millions = $millions;

    // Trillions
    if ($millions ≥ 1_000_000) {
      $this->formatted = number_format(
        $this->millions / 1_000_000, 2
      ) . 'T';
    }

    // Billions
    if ($millions < 1_000_000 && $millions ≥ 1_000) {
      $this->formatted = number_format(
        $this->millions / 1_000, 1
      ) . 'B';
    }

    // Millions
    if ($millions < 1_000) {
      $this->formatted = number_format($this->millions) . 'M';
    }
  }
```

```
  public static function from(int $millions): self
  {
    return new self($millions);
  }
}
```

We need to check the value of `$millions` and do the appropriate division and use the right suffix.

The cast is almost identical to `PriceCast`:

```
namespace App\Models\Casts;

class MarketCapCast implements CastsAttributes
{
  public function get($model, $key, $value, $attributes)
  {
    return MarketCap::from($value);
  }

  public function set($model, $key, $value, $attributes)
  {
    return $value;
  }
}
```

Once again, we don't need to do anything in `set`. The last thing is to use this cast:

```php
namespace App\Models;

class Company extends Model
{
  use HasFactory;

  protected $guarded = [];

  protected $casts = [
    'price_per_share' => PriceCast::class,
    'market_cap' => MarketCapCast::class,
  ];
}
```

I won't list the other `Cast` classes because all of them are the same. You can check them out in the repository.

Millions

This value object is pretty simple:

```
namespace App\ValueObjects;

class Millions
{
  public readonly int $value;
  public readonly int $millions;
  public readonly string $formatted;

  public function __construct(int $millions)
  {
    $this->value = $millions * 1_000_000;

    $this->millions = $millions;

    $this->formatted = number_format($this->millions, 0, ',');
  }

  public static function from(int $millions): self
  {
    return new self($millions);
  }
}
```

There are three properties:

- `value` contains the raw number as an integer.
- `millions` contains the number expressed in millions.
- `formatted` contains the formatted number, something like `192,557`

As JSON:

```
"revenue": {
  "2022": {
    "value": 192557000000,
    "millions": 192557,
    "formatted": "192,557"
  }
}
```

`Millions` is used in the `IncomeStatement` model, and this is where we benefit from using Eloquent `Casts`:

```php
namespace App\Models;

class IncomeStatement extends Model
{
  use HasFactory;

  protected $guarded = [];

  protected $casts = [
    'revenue' => MillionsCast::class,
    'cost_of_revenue' => MillionsCast::class,
    'gross_profit' => MillionsCast::class,
    'operating_expenses' => MillionsCast::class,
    'operating_profit' => MillionsCast::class,
    'interest_expense' => MillionsCast::class,
    'income_tax_expense' => MillionsCast::class,
    'net_income' => MillionsCast::class,
    'eps' => PriceCast::class,
  ];
}
```

Margin

It's also a fairly simple class:

```
namespace App\ValueObjects;

class Margin
{
  public readonly float $value;
  public readonly string $formatted;
  public readonly Millions $top_line;
  public readonly Millions $bottom_line;

  public function __construct(
    float $value,
    Millions $topLine,
    Millions $bottomLine
  ) {
    $this->value = $value;

    $this->top_line = $topLine;

    $this->bottom_line = $bottomLine;

    $this->formatted = number_format($value * 100, 2) . '%';
  }
```

```
  public static function make(
    float $value,
    Millions $topLine,
    Millions $bottomLine
  ): self {
    return new self($value, $topLine, $bottomLine);
  }
}
```

This shows another great feature of value objects: they can be nested. In this example, the `top_line` and `bottom_line` attributes are `Millions` instances. These numbers describe how the margin is calculated. For example, the gross margin is calculated by dividing the revenue (top line) by the gross profit (bottom line). This will look like this in JSON:

```
"gross_margin": {
  "2022": {
    "value": 0.68,
    "formatted": "68.00%",
    "top_line": {
      "value": 192557000000,
      "millions": 192557,
      "formatted": "192,557"
    },
    "bottom_line": {
      "value": 132345000000,
      "millions": 132345,
      "formatted": "132,345"
    }
  }
}
```

However, if you take a look at the `make` method, you can see we expect two additional parameters: `$topLine` and `$bottomLine`. This means we can use this object like this:

```
$company = Company::first();

$incomeStatement = $company->income_statements()
  ->where('year', 2022)
  ->first();

$metrics = $company->metrics()->where('year', 2022)->first();

$grossMargin = Margin::make(
  $metrics->gross_margin,
  $incomeStatement->revenue,
  $incomeStatement->gross_profit,
);
```

Since we are using Eloquent `Casts` we need the revenue and gross profit (in this specific example) in the `MarginCast` class. We can do something like this:

```php
namespace App\Models\Casts;

class MarginCast implements CastsAttributes
{
  /**
   * @param Metric $model
   */
  public function get($model, $key, $value, $attributes)
  {
    $incomeStatement = $model
      ->company
      ->income_statements()
      ->where('year', $model->year)
      ->first();

    [$topLine, $bottomLine] = $model->getTopAndBottomLine(
      $incomeStatement,
      $key,
    );

    return Margin::make($value, $topLine, $bottomLine);
  }

  public function set($model, $key, $value, $attributes)
  {
    return $value;
  }
}
```

As you can see, the model, in this case, is a `Metric` model (this is where the cast will be used) so we can query the appropriate income statement for the same year. After that, we need a method that can return the top and bottom line for a particular metric:

```
namespace App\Models;

class Metric extends Model
{
  public function getTopAndBottomLine(
    IncomeStatement $incomeStatement,
    string $metricName
  ): array {
    return match ($metricName) {
      'gross_margin' => [
        $incomeStatement->revenue,
        $incomeStatement->gross_profit
      ],
      'operating_margin' => [
        $incomeStatement->revenue,
        $incomeStatement->operating_profit
      ],
      'profit_margin' => [
        $incomeStatement->revenue,
        $incomeStatement->net_income
      ],
    };
  }
}
```

This method simply returns the right items from the income statement based on the metric. The logic is quite simple, but it's much more complicated than the other ones, so I recommend you to check out the source code and open these classes.

You may be asking: "Wait a minute... We are querying companies and income statements in the MarginCast for every attribute??? That's like 10 extra queries every time we query a simple Metric, right?"

Good question! The answer is: nope. These casts are lazily executed. This means the `get` function will only be executed when you actually access the given property. But as you might already guess we'll access every property in a resource, so a bunch of extra queries will be executed. What can we do about it?

- Eager load relationships when querying a metric. This will prevent us from running into N+1 query problems.
- Cache the income statements. After all, they are historical data, updated once a year. This will also prevent extra queries.
- If performance is still an issue, you can drop the whole `MarginCast` class, and use the object in the resource directly. In this case, you have more flexibility. For example, you can query every important data in one query, and only interact with collections when determining the top and bottom line values.

PeRatio

After all of these complications, let's see the last and probably most simple VO:

```
namespace App\ValueObjects;

class PeRatio
{
  public readonly string $value;

  public function __construct(int $peRatio)
  {
    $this->value = number_format($peRatio / 100, 2);
  }

  public static function from(int $peRatio): self
  {
    return new self($peRatio);
  }
}
```

This class can also be used to cover other ratio-type numbers, but right now PE is the only one, so I decided to call the class `PeRatio`.

Income Statement Summary

Now that we have all the value objects, we can move on to the resource. Our goal is to get a summary view of the income statements of the company. This is the JSON structure:

```
"data": {
  "years": [
    2022,
    2021
  ],
  "items": {
    "revenue": {
      "2022": {
        "value": 386017000000,
        "millions": 386017,
        "formatted": "386,017"
      },
      "2021": {
        "value": 246807000000,
        "millions": 246807,
        "formatted": "246,807"
      }
    }
  }
}
```

There are at least two ways we can approach this problem:

- A more "static" approach
- And a more "dynamic" one

By "dynamic," I mean something like this:

```
class IncomeStatementResource
{
  public $preserveKeys = true;

  public function toArray(Request $request)
  {
    $data = [];

    // $this is a Company
    $data['years'] = $this->income_statements->pluck('year');

    foreach ($this->income_statements as $incomeStatement) {
      foreach ($incomeStatement->getAttributes() as $attribute
=> $value) {
        $notRelated = [
          'id', 'year', 'company_id',
          'created_at', 'updated_at',
        ];

        if (in_array($attribute, $notRelated)) {
          continue;
        }

        Arr::set(
          $data,
          "items.{$attribute}.{$incomeStatement->year}",
          $incomeStatement->{$attribute}
        );
```

```
        }
      }

      return $data;
    }
}
```

Are you having a hard time understanding what's going on? It's not your fault! It's mine. This code sucks. I mean, it's very "dynamic" so it'll work no matter if you have four columns in the `income_statements` or 15. But other than that it seems a bit funky to me. Moreover, it has no "real" form, so it's very weird to put it in a resource.

Don't get me wrong, sometimes you just need solutions like this. But an income statement has a finite amount of items (columns), and it's not something that is subject to change.

Let's see a more declarative approach:

```
namespace App\Http\Resources;

class IncomeStatementsSummaryResource extends JsonResource
{
  public $preserveKeys = true;

  public function toArray($request)
  {
    // $this is a Collection<IncomeStatement>
    $years = $this->pluck('year');

    return [
      'years' => $years,
      'items' => [
```

```
            'revenue' => $this->getItem(
              'revenue',
              $years
            ),
            'cost_of_revenue' => $this->getItem(
              'cost_of_revenue',
              $years
            ),
            'gross_profit' => $this->getItem(
              'gross_profit',
              $years
            ),
            'operating_expenses' => $this->getItem(
              'operating_expenses',
              $years
            ),
            'operating_profit' => $this->getItem(
              'operating_profit',
              $years
            ),
            'interest_expense' => $this->getItem(
              'interest_expense',
              $years
            ),
            'income_tax_expense' => $this->getItem(
              'income_tax_expense',
              $years
            ),
            'net_income' => $this->getItem(
              'net_income',
```

```
            $years
          ),
          'eps' => $this->getItem(
            'eps',
            $years
          ),
        ]
      ];
    }

    /**
     * @return array<int, int>
     */
    private function getItem(
      string $name,
      Collection $years
    ): array {
      $data = [];

      foreach ($years as $year) {
        $data[$year] = $this
          ->where('year', $year)
          ->first()
          ->{$name};
      }

      return $data;
    }
}
```

Can you see the difference? It's easy to understand, readable has a real form, and does not require more code at all (all right, in this PDF it seems much longer, but in the repository, each item is one line). However, it's called `IncomeStatementsSummaryResource`, and there's a reason why. This resource requires a `Collection<IncomeStatement>` so it can be used like this:

```
namespace App\Http\Controllers;

class IncomeStatementController extends Controller
{
  public function index(Company $company)
  {
    return IncomeStatementsSummaryResource::make(
      $company->income_statements
    );
  }
}
```

We pass all the income statements of a company as a Collection. So this line in the resource won't run additional queries:

```
// $this->where() is a Collection method
$data[$year] = $this->where('year', $year)->first()->{$name};
```

The last important thing is this line here:

```
public $preserveKeys = true;
```

Without this Laravel will override the array keys and it'll convert the years to standard zero-based array indices:

```
"data": {
  "years": [
    2022,
    2021
  ],
  "items": {
    "revenue": [
      {
        "value": 386017000000,
        "millions": 386017,
        "formatted": "386,017"
      },
      {
        "value": 246807000000,
        "millions": 246807,
        "formatted": "246,807"
      }
    ]
  }
}
```


As you can see the year-based object becomes a JSON array. This is why I used the `$preserveKeys` property from the parent `JsonResource` class.

Metrics Summary

The metrics summary API is basically the same as the income statement. So not surprisingly the Resource looks almost the same:

```
namespace App\Http\Resources;

class MetricsSummaryResource extends JsonResource
{
  public $preserveKeys = true;

  public function toArray($request)
  {
    $years = $this->pluck('year');

    return [
      'years' => $years,
      'items' => [
        'gross_margin' => $this->getItem(
          'gross_margin',
          $years
        ),
        'operating_margin' => $this->getItem(
          'operating_margin',
          $years
        ),
        'profit_margin' => $this->getItem(
          'profit_margin',
          $years
        ),
```

```
            'pe_ratio' => $this->getItem(
              'pe_ratio',
              $years
            ),
          ]
        ];
    }

    private function getItem(
        string $name,
        Collection $years
    ): array {
        $data = [];

        foreach ($years as $year) {
            $data[$year] = $this
                ->where('year', $year)
                ->first()
                ->{$name};
        }

        return $data;
    }
}
```

Can be used like this:

```php
namespace App\Http\Controllers;

class MetricController extends Controller
{
  public function index(Company $company)
  {
    return MetricsSummaryResource::make($company->metrics);
  }
}
```

Conclusion

It was a longer exclusive, I know. Give it some time, maybe read it again later.

Value objects are awesome, in my opinion! I almost use them in every project, no matter if it's old, new, DDD, or not DDD, legacy, or not. It's pretty easy to start using them, and you'll have a very high-level, declarative codebase.

I often got the question: "what else can be expressed as a value object?" Almost anything, to name a few examples:

- Addresses. In an e-commerce application where you have to deal with shipping, it can be beneficial to use objects instead of strings. You can express each part of an address as a property:
  - City
  - ZIP code
  - Line 1
  - Line 2
- Numbers and percents. As we've seen.
- Email addresses.
- Name. With parts like first, last middle, title
- Any measurement unit, such as weight, temperature, distance
- GPS coordinates.
- EndDate and StartDate. They can be created from a Carbon but ensure that a StartDate is always at 00:00:00 meanwhile an EndDate is always at 23:59:59.
- Any other application-specific concepts.

Static Analysis

In general, a static analysis tool helps you avoid:

- Bugs
- Too complex methods and classes
- Lack of type-hints
- Poorly formatted code

There are an infinite amount of tools out there, but in the following pages, I'd like to show my three favorite tools.

phpinsights

This is my number #1 favorite tool. It gives you an output like this:

It scores your codebase on a scale from 1..100 in four different categories:

- Code. It checks the general quality of your code. It doesn't involve any style of format, but only quality checks such as:
  - Unused private properties
  - Unnecessary final modifiers
  - Unused variables
  - Correct switch statement

    - And much more
- Complexity. In my opinion, **this is the most critical metric**. It simply checks how complicated a class is, using cyclomatic complexity. It's a fancy way of saying: how many different execution paths exist in a function. Or put it simply: how many if-else or switch statements do you have in one function or class. By default, it raises an issue if the average is over 5.
- Architecture. It checks some general architectural stuff, such as:
    - Method per class limit
    - Property per class limit
    - Superfluous interface or abstract class naming
    - Function length
    - And so on
- Style. It checks some style-related rules, for example:
    - No closing PHP tag at the end of files
    - There's a new line at the end of files
    - Space after cast

By default, it doesn't require any configuration at all. Of course, if you don't want to use some rules, you can disable them. Check out the documentation for more information. phpinsights can be run by issuing this command:

```
./vendor/bin/phpinsights analyse
```

It gives you an excellent summary and an interactive terminal where you can see every issue. But it also ships with a 'non-interactive' mode, and you can also define the minimum scores you want to have:

```
./vendor/bin/phpinsights --no-interaction --min-quality=80 --
min-complexity=90 --min-architecture=70 --min-style=75
```

The `--no-interaction` flag means that the terminal window does not expect any input; it just gives you the summary and every error message. The other `--min-xy` flags make it possible to define the minimum scores for each category. For example, if the complexity score drops below 90%, the command will yield a non-zero output and an error message. **The minimum complexity score is always 90% for me.**

larastan

Larastan is a Laravel specific tool built on top of phpstan. These two use the same configuration format and rule system. It ships with a default ruleset (very strict) and has a config parameter called `level`. This parameter determines how strict it is and how many rules are applied. If you want to learn more about these rules, check the documentation.

The config file looks like this:

```
includes:
  - ./vendor/nunomaduro/larastan/extension.neon

parameters:
  paths:
    - app
    - src

  level: 5
  ignoreErrors:
    - '#PHPDoc tag @var#'
  excludePaths:
    - 'app/Http/Kernel.php'
    - 'app/Console/Kernel.php'
    - 'app/Exceptions/Handler.php'
  checkMissingIterableValueType: false
  noUnnecessaryCollectionCallExcept: ['pluck']
```

Important values are:

- We need to use src to scan all of the domains in the `paths`.

- `level` goes from 1..9. Basically, you need to experiment with what level is best for you, but here's my general rule:
  - Legacy project: start with 1. In my opinion, you have no other options if the static analysis is new to the project.
  - Fresh application: somewhere between 4 and 6, but it heavily depends on the team and the project.
  - Never reach for level 9. Seriously, it gets pretty hard above level 5. My all-time best was level 7, and I was dying during the process. It's like a tough game where you cannot beat the final boss.
- Under the `excludePaths` you can list files or directories that you want to exclude. Sometimes I exclude default Laravel files such as the ones above.

You can browse the full config in the `phpstan.neon` file (root directory of the sample app).

You can run the rules with this command:

```
./vendor/bin/phpstan analyse
```

Laracheck

I'm a bit biased toward Laracheck because it's my product so this chapter is going to be an evil sales pitch😈

It's a code review tool available on your GitHub repos and it performs the following check when you open a PR:

N+1 query detection

Anytime you write a foreach loop or call a Collection method it will look for potential N+1 problems.

Here are some examples that qualify as a problem:

- You access a relationship that is not eager-loaded either in the body of the current function (using *with() or load()*) or in the model itself (using the *$with* property).
- You call DB functions in the loop such as *DB::table()*
- You call static Model functions in the loop such as *Product::find()*
- You call Model functions in the loop such as *$product->save()*

Incorrect dependencies

There are different layers in every Laravel application. Layers such as HTTP, Business Logic, Database, etc. Each layer has its own dependencies. For example, the database layer should not depend on the HTTP layer. If it does, Laracheck will show you a warning.

Here are what counts as an incorrect dependency:

This class	Depends on these
Model	HTTP, Job, Command, Auth
Job	HTTP
Command	HTTP
Mail/Notification	HTTP, Job, Command
Service	HTTP
Repository	HTTP, Job, Command

As you can see, one of the most common issues I used to face is when a class depends on an HTTP-related class. Such as a model using a request. It's a bad practice in my opinion because we couple the transportation layer (HTTP) to the database layer. One of the problems it causes is the lack of reusability. For example, you cannot use this model function from a command or job because they don't have requests.

The inner layers of your application (such as models) should not depend on outer layers (such as HTTP).

Complex data objects

There are some typical classes that should not contain too much business logic since their main purpose is to hold data. These classes are:

- Resources
- Requests
- DataTransferObjects (DTO)
- Value Objects
- Mail
- Notification

If you have a class that contains too much business logic, Laracheck will warn you. "Too much" means that the cyclomatic complexity of the class is larger than 3.

`env()` calls outside of config files

In Laravel, it's a best practice to use `env('MY_ENV_VALUE')` calls only in config files. There are two reasons.

Often config values are cached in production environment using the `php artisan config:cache` command. If you don't know about this command, you should consider using it. It'll cache every config file into memory. So whenever you use them with `config('app.my_value')` it'll retrieve the value from memory instead of touching the `.env` file on the disk.

If you have `env()` calls in your code (outside of config files), this config caching can break your production environment! Or at least it can cause bugs.

The other reason is that `config` values can be "mocked" in tests pretty easily. All you have to do is this:

```
class ListProductsTest extends TestCase
{
  use RefreshDatabase;

  /** @test */
  public function it_should_use_the_default_page_size()
  {
    config(['app.default_page_size' => 10]);

    $products = $this->getJson(route('products.index'))
      ->json('data');

    $this->assertCount(10, $products);
  }
```

```
  /** @test */
  public function it_should_return_n_products()
  {
    $products = $this->getJson(route('products.index', [
      'per_page' => 20
    ]))
      ->json('data');

    $this->assertCount(20, $products);
  }
}
```

This way you can test multiple config values, you can easily turn on and off feature flags, and so on.

I'm not gonna go into more detail about the other checks but here's a list of the most important ones:

- N+1 query detection
- Missing whenLoaded() calls
- Missing DB index in migration
- Missing down method in migration
- Missing foreign key in migration
- Missing authorization in request
- Validation in controller
- Missing tests
- Missing ENV variable
- Missing/changed composer lock file
- env() call outside of config files
- Forgotten cache keys
- Incorrect dependencies
- Complex data object

- Custom Checks

Try out Laracheck for free.

deptrac

This tool helps you to clean up your architecture. In the book, I used several classes. The important ones are:

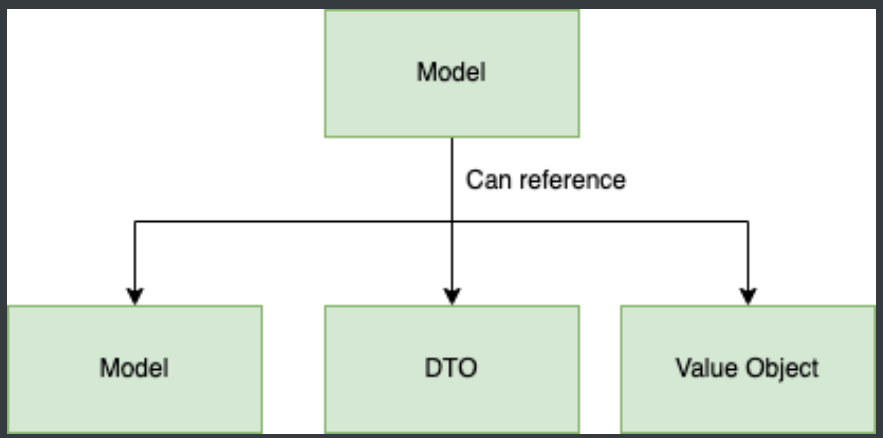
- Controllers
- Action
- ViewModels
- Builders
- Models
- DTOs

Each class has its purpose. For example, I don't want a controller to start implementing business logic. It has three responsibilities, in my opinion:

- Accepting a request.
- Calling the necessary methods from another class.
- Returning a response.

Another important rule is to keep the models lightweight. If the models are sending notifications, dispatching jobs, or calling APIs, it's a bad design, in my opinion.

If you think about it, these architectural rules can be enforced by simply defining which class a Model can reference, right? Something like that:



deptrac does precisely that. If these rules are not followed, and a Model uses a Job deptrac will throw a huge red screen in your face.

We can configure these rules in the `deptrac.yaml` file:

```
parameters:
  paths:
    - ./app
    - ./src
  exclude_files:
    - '#.*test.*#'
    - '#.*Factory\.php$#'
  layers:
    - name: Action
      collectors:
        - type: className
          regex: .*Actions\\.*
```

This tells deptrac that the project has a layer called Action, and the files can be collected using this regex `.*Actions\\.*` It means that every file inside the `Actions` folder is an action class. After the layers are created, we can define the rulesets:

```
ruleset:
    Controller:
      - Action
      - ViewModel
      - Model
      - DTO
      - ValueObject
    Action:
      - Event
      - Model
      - DTO
```

```
      - Builder
      - ValueObject
    Model:
      - Builder
      - Model
      - DTO
      - ValueObject
    DTO:
      - Model
      - DTO
      - ValueObject
    ValueObject:
      - ValueObject
```

This means that a model in the project can only use:

- Other models.
- Query builders.
- DTOs.
- Value objects.

If anything else is referenced inside a model, it will throw an error. You can find the config in the `deptrac.yaml` inside the sample application. If you want to run it, just run this command:

```
./vendor/bin/deptrac analyse
```

So these are my favorite static analysis tools. But the true power comes when you integrate these tools into your CI/CD pipeline.
I highly recommend using this tool in new projects. It requires only 15-30 minutes to set up, but it provides value for the next 3-5 years. Also, if you have a legacy project that you want to clean up, this package can be really helpful!

Working with OS Processes

From time to time it can happen that we need to run some external processes. I'm talking about things that don't have an SDK or a PHP-related function. In the last year, I had to interact with two of those:

- git
- terragrunt/terraform

These are programs that are installed on the host machine (or in the Dockerfile) but don't have an SDK or a function such as `file_get_contents()`.

In these cases, we can use the amazing Symfony component called Process:

```
use Symfony\Component\Process\Process;

$process = new Process(
  ['git', 'commit', '-m', 'Commit message']
);

$process->run();
```

The constructor of the `Process` class takes an array. Each element is a part of the command as you can see.

To get the output of the process we can use the `getOutput` method:

```
use Symfony\Component\Process\Process;

$process = new Process(
  ['git', 'commit', '-m', 'Commit message']
);

$process->run();

$output = $process->getOutput();
```

It returns a string and it contains the exact output from the `git` process. This is the same that you see in your terminal.

To handle errors we can use the `isSuccessful` method:

```
use Symfony\Component\Process\Process;

$process = new Process(
  ['git', 'commit', '-m', 'Commit message']
);

$process->run();

if (!$process->isSuccessful()) {
  throw new ProcessFailedException($process);
}

return $process->getOutput();
```

These are the basics of the `Process` component. Now we have everything to create a general `GitService` that can run anything:

```
namespace App\Services;

use Symfony\Component\Process\Exception\ProcessFailedException;
use Symfony\Component\Process\Process;

class GitService
{
  /**
   * @param array $command
   * @return string
   * @throws ProcessFailedException
   */
  public function runCommand(array $command): string
  {
    $process = new Process($command);

    $process->run();

    if (!$process->isSuccessful()) {
      throw new ProcessFailedException($process);
    }

    return $process->getOutput();
  }
}
```

And we can use it like this:

```
public function index(GitService $git)
{
  $git->runCommand(
    ['git', 'commit', '-m', 'Commit message']
  );
}
```

If you look at this class it's a little bit weird. We call it `GitService` but is has no git-specific logic. Even worse, we need to pass the word `git` as an argument. The problem is that `GitService` is not a real `GitService` at this point. It's just a generic process wrapper or something like that.

So let's make it more like a `GitService` :

```
class GitService
{
  public function pull(): string
  {
    return $this->runCommand(['git', 'pull']);
  }

  public function commit(string $message): string
  {
    $this->runCommand(['git', 'add', '--all']);

    return $this->runCommand(
      ['git', 'commit', '-m', $message]
    );
  }
```

```
  public function push(): string
  {
    return $this->runCommand(['git', 'push']);
  }
}
```

Now the usage looks like this:

```
public function index(GitService $git)
{
  $git->commit('Add git service');
}
```

Now it's much better. Each `git` command has its own method which is a good practice in my opinion. After these changes, we don't really want to access the `runCommand` method outside of this class so it can be private.

Another minor problem you might notice is that we are using arrays just because Symfony Process expects us to pass arrays:

```
$this->runCommand(['git', 'commit', '-m', $message]);
```

Yeah, it looks a bit weird, so we can refactor it to accept a string instead:

```
$this->runCommand("git commit -m '$message'");
```

However, parsing the command becomes tricky. For example, consider this:

```
$this->runCommand("git commit -m 'Add git service'");
```

We want to make an array from this string that looks like this:

```
[
  "git",
  "commit",
  "-m",
  "Add git service",
]
```

We can use the `explode` function, but if the message contains spaces the result looks like this:

```
[
  "git",
  "commit",
  "-m",
  "'Add",
  "git",
  "service'",
]
```

And the command will fail. So using strings might look 7% better, it just doesn't worth the potential bugs and complexity in my opinion.

Another great feature of the `Process` class is that it can give us real-time output. It's pretty useful when you're working with long-running processes (such as `terragrunt init` or `apply`). To get logs as they come, we can use a loop:

```php
$process = new Process(['terragrunt', 'apply']);

$process->start();

foreach ($process as $type => $data) {
  if ($process::OUT === $type) {
    echo "Info: " . $data;
  } else {
    echo "Error: " . $data;
  }
}
```

So if you ever need to work with OS processes, just forget about `exec` and go with Symfony Process. It's a great component!

Custom Query Builders

In bigger projects, we often struggle with models that have too much business logic in them. Fortunately, you can build your own query builder classes to make your models a bit leaner.

Let's say we have an Article model:

```
class Article extends Model
{
  use HasFactory;

  protected $protected = [];

  protected $casts = [
    'published_at' => 'datetime',
  ];

  public function author(): BelongsTo
  {
    return $this->belongsTo(User::class);
  }

  public function ratings(): HasMany
  {
    return $this->hasMany(Rating::class);
  }
}
```

Any time you write something like that:

```
Article::where('title', 'My Awesome Article')->get();
```

You interact with the `Illuminate\Database\Eloquent\Builder` class under the hood:

```
custom-query-builders tink

Psy Shell v0.11.10 (PHP 8.1.12 — cli) by Justin Hileman
> Article::where('title', 'My Awesome Article')
[!] Aliasing 'Article' to 'App\Models\Article' for this Tinker session.
= Illuminate\Database\Eloquent\Builder {#3692}

>
```

The base `Model` class in Laravel has a `newEloquentBuilder` method:

```
/**
 * Create a new Eloquent query builder for the model.
 *
 * @param  \Illuminate\Database\Query\Builder  $query
 * @return \Illuminate\Database\Eloquent\Builder|static
 */
public function newEloquentBuilder($query)
{
  return new Builder($query);
}
```

If you check the base `Model` class it doesn't have methods like `where`, `whereBetween`, or anything like that. All of these functions come from the `Builder` class. When you write your query, for example, `Article::where(...)` Laravel first calls the `newEloquentBuilder` method. It returns a `Builder` instance which has functions such as

`where`.

Since the `newEloquentBuilder` method is defined in the `Model` class, we can override it:

```
use App\Builders\ArticleBuilder;

class Article extends Model
{
  use HasFactory;

  protected $protected = [];

  protected $casts = [
    'published_at' => 'datetime',
  ];

  public function newEloquentBuilder($query): ArticleBuilder
  {
    return new ArticleBuilder($query);
  }
}
```

And we can create a class called `ArticleBuilder` that extends the base `Builder` class:

```php
<?php

namespace App\Builders;

use App\Models\User;
use Illuminate\Database\Eloquent\Builder;

class ArticleBuilder extends Builder
{
}
```

Now if we start writing a query we get an `ArticleBuilder` instance:

```
custom-query-builders tink
Psy Shell v0.11.10 (PHP 8.1.12 — cli) by Justin Hileman
> Article::where('title', 'My Awesome Article')
[!] Aliasing 'Article' to 'App\Models\Article' for this Tinker session.
= App\Builders\ArticleBuilder {#3692}

>
```

So what we can do with this new class?

Scopes

Did you know that model scope is just syntactic sugar around query builders? Here's how you can use them without magic:

```
class ArticleBuilder extends Builder
{
  public function wherePublished(): self
  {
    return $this->where('published_at', '≤', now());
  }

  public function whereAuthor(User $user): self
  {
    return $this->where('author_id', $user->id);
  }
}
```

I like to start every scope with where because it seems more expressive in a query. The important thing is that you have to return an `ArticleBuilder` instance from every method since we want to chain these methods. Notice that there is no get() or all() or anything like that after the where() calls.

These scopes can be used as if they were in the model:

```
class ArticleController
{
  public function myArticles(Request $request)
  {
    return Article::query()
      ->whereAuthor($request->user)
      ->wherePublished()
      ->orderBy('created_at', 'desc')
      ->get();
  }
}
```

In the `ArticleBuilder` class you have no limitations, you can build any query you want. Here's one with some where groups:

```
public function whereContains(string $searchTerm): self
{
  return $this->where(function ($query) use ($searchTerm) {
    $query->where('title', 'LIKE', "%$searchTerm%")
      ->orWhere('summary', 'LIKE', "%$searchTerm%");
  });
}
```

Queries

Of course, you don't have to write only scope-like functions that are chainable. Here's a standard query:

```
class ArticleBuilder extends Builder
{
  /**
   * @return Collection<Article>
   */
  public function getOldArticles(): Collection
  {
    return $this
      ->where('created_at', '≤', now()->subYears(5))
      ->get();
  }
}
```

This method simply returns a list of Articles just as a regular model query would be:

```
$oldArticles = Article::getOldArticles();
```

We can also work with relationships the same we used to:

```
class ArticleBuilder extends Builder
{
  public function orderByRatings(): self
  {
    return $this
      ->withAvg('ratings', 'rating')
      ->orderByDesc('ratings_avg_rating');
  }
}
```

The usage is simple:

```
$articles = Article::query()
  ->wherePublished()
  ->orderByRatings()
  ->get();
```

All the relationship aggregate functions are available such as `withCount` or `withAvg`.

One important thing though. If you want to write a method that manipulates a concrete Article record, you need to do this:

```
class ArticleBuilder extends Builder
{
  public function publish(): void
  {
    if ($this->model->published_at) {
      return;
    }

    $this->model->published_at = now();

    $this->model->save();
  }
}
```

So in a `Builder` you can access the model instance as `$this->model`. The `publish` function can be used in a straightforward way:

```
$article = Article::first();

$article->publish();
```

Separation

Custom query builders are a great way to make your models smaller and simpler. However, all we did in this example, is we moved code from the `Article` class to the `ArticleBuilder` class. As you can imagine, in the long term the result will be the same, but in this case, the `ArticleBuilder` will become a huge class.

Another approach to solving this problem is this:

- Write your "static" queries and scopes in `Builder` classes
- Write your non-static methods in `Models`

By "static" I mean functions that don't interact with one particular record such as these functions:

```
class ArticleBuilder extends Builder
{
  public function whereAuthor(User $user): self
  {
    return $this->where('author_id', $user->id);
  }

  public function orderByRatings(): self
  {
    return $this
      ->withAvg('ratings', 'rating')
      ->orderByDesc('ratings_avg_rating');
  }
}
```

And by non-static I mean the `publish` and `unpublish` methods:

```
class Article extends Model
{
  public function publish(): void
  {
    if ($this->published_at) {
      return;
    }

    $this->published_at = now();

    $this->save();
  }

  public function unpublish(): void
  {
    $this->published_at = null;

    $this->save();
  }
}
```

In usage there's no difference:

```
$articles = Article::query()
  ->whereAuthor($request->user())
  ->wherePublished();

foreach ($articles as $article) {
  $article->unpublish();
}
```

Or another approach would be to write your queries inside `Builder` classes and use `Actions` or `Services` to handle user stories. This way, your models only represent a record in the database without any business logic.

Final Words

Thank you very much for reading this book! I hope you liked it. If you have any question just send me an e-mail and I try to reply as soon as possible.

If you want to learn more about Laravel and software engineering in general, check out my blog. I also published other books:

- Domain-Driven Design with Laravel
- Microservices with Laravel
- Test-Driven APIs with Laravel and Pest