

SOLID Principles

- Single-Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Building a Dashboard

- Data modeling
- Gumroad Service
 - Get Products
 - Get Sales
- Sync With Gumroad
 - Sync Gumroad Products
 - Sync Gumroad Sales
- Dashboard

Handling Discounts and Coupons

- Data Modeling
- Expired Coupons
- Inactive Coupons
- Coupon Conditions
- Coupon types

Working with 3rd Parties

- One Service
- Separate Requests
- Transporter by JustSteveKing
- Laravel Forge SDK

Custom Fields

- Data Modeling
- User API
- User Field API

Final Words

SOLID Principles

First, let's discuss what SOLID stands for:

- Single-responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Sounds scientific, isn't it? But it's all just marketing. In reality, this is the most simple thing in the universe. It's a set of principles mainly focusing on object-oriented programming made popular by Robert C. Martin. Now let's see what are these principles.

Single-Responsibility Principle

Each class should have only one reason to change.

It's kind of hard to define what a "reason" might be and it causes some confusion but usually, it's related to roles. Users have different roles. For example, let's say we're working on an app that is being used by financial experts. Users want reports. It's obvious that an accountant wants to see completely different reports and charts than a CFO. Both of them are reports, but they are being used in different roles. So it's probably a good idea not to mix the code in one `Report` class. This one class will change for different reasons.

Another, maybe more obvious example is data and its representation. Usually, they change for different reasons. Hence it's probably a safe bet to decouple the query layer from the representation layer. Which is the defacto industry standard nowadays, and one of the reasons why API+SPA is so popular. But it wasn't always the case. But we can take a step back because there are still countless examples when developers mix these two things in Laravel.

Consider this hypothetical (and oversimplified) example:

```
class UserResource extends JsonResponse
{
    public function toArray($request)
    {
        $mostPopularPosts = $user->posts()
            ->where('like_count', '>', 50)
            ->where('share_count', '>', 25)
            ->orderByDesc('visits')
            ->take(10)
            ->get();
    }
}
```

```
return [  
    'id' => $this->id,  
    'full_name' => $this->full_name,  
    'most_popular_posts' => $mostPopularPosts,  
];  
}  
}
```

I couldn't count how many times I see something like that. This is a modern example of mixing the data and the representation layer in one class. When we see a legacy project where there is a single PHP file with HTML, PHP, and MySQL in it, we cry in pain. Of course, this resource is much better than that, but in fact, it has similar issues:

- The array is similar to HTML. It's the representation of the data.
- The Eloquent query is similar to MySQL. It's the query layer.
- And the whole class is in PHP.

So after all, we're mixing a lot of things in just 20 lines of code.

"All right, but it's all theory. What's the big deal about this?" Here is some problems that might occur:

- The PM says: "can we please change the definition of 'most popular posts'?" After a request like this, I'd not think that I need to go to the `UserResource` class. It's just not logical. This class shouldn't be changed because of that request.
- You probably need this resource in a lot of places all around your application. But information such as most popular posts is typically shown on only a few pages. So it's wasteful.
- This simple query can be the origin of N+1 queries and other performance issues.

Fortunately, the fix is pretty easy:

```

class UserResource extends JsonResource
{
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'full_name' => $this->full_name,
            'most_popular_posts' => $this->when(
                $request->most_popular_posts,
                $this->mostPopularPosts,
            ),
        ];
    }
}

class User extends Posts
{
    /**
     * @return Collection<Post>
     */
    public function mostPopularPosts(): Collection
    {
        return $this->posts()
            ->where('like_count', '>', 50)
            ->where('share_count', '>', 25)
            ->orderByDesc('visits')
            ->take(10)
            ->get();
    }
}

```

Of course in this example, I don't assume anything about the overall architecture of the project. You can write services, actions, query builders, scopes, repositories, or anything you like. You can argue that this query should be in the `Post` model.

We can do something like that:

```
Post::mostPopularBy($user);
```

Or using a scope:

```
$user->posts()->mostPopularOnes();
```

I also think that it's a better solution. If a PM says: "can we please change the definition of 'most popular posts'?" I know I need to go to the `Post` model. And of course, usually, the `User` is the first one that is going into "legacy" mode. After six months. So you can see that a simple query like this can cause some confusion and debate. Or even (often) religious wars.

This is why I prefer using single-use-case actions. And I think this is why they are getting more and more popular in the Laravel community. It looks like this:

```

class UserResource extends JsonResource
{
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'full_name' => $this->full_name,
            'most_popular_posts' => $this->when(
                $request->most_popular_posts,
                GetMostPopularPosts::execute($user),
            ),
        ];
    }
}

class GetMostPopularPosts
{
    /**
     * @return Collection<Post>
     */
    public static function execute(User $user): Collection
    {
        return $user->posts()
            ->where('like_count', '>', 50)
            ->where('share_count', '>', 25)
            ->orderByDesc('visits')
            ->take(10)
            ->get();
    }
}

```

You can also use non-static functions, or invokable classes the choice is yours. All of them are great and testable so I think it's just a matter of preference. But can you see how much we improved the "architecture" from the single-responsibility point-of-view?

Now we have two well-defined classes:

- `UserResource` is responsible only for the representation and it has one reason to change.
- `GetMostPopularPosts` is responsible only for the query and it has one reason to change.

Here are some typical indicators that you're breaking SRP:

- Database queries in simple "data" or representation classes such as requests, responses, DTOs, value objects, mails, and notifications. Anytime you write business logic into these classes it can be a bad practice.
- Dispatching jobs or commands from models. Usually, you don't want to couple these things together. It's better to use an event or dispatch the job from a controller or use an action. Models should not be orchestrator classes that are starting long-running processes.
- Incorrect dependencies. It breaks a lot of other principles but is usually also a red flag in terms of SRP. By incorrect dependency, I mean, when a Model uses an HTTP request or response. In this case, you're coupling a transportation layer (HTTP) to a data layer (model). These are some examples I consider "incorrect" dependencies:

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

Of course, these are just overgeneralized examples. Usually, it depends on your exact project/class.

Open-Closed Principle

A class should be open for extension but closed for modification

It sounds weird, I know. Please don't check out the Wikipedia page because it gets even weirder. So let me show you an example.

Let's say we're working on a social application. It has users, posts, comments, and likes. Users can like posts, so you implement this feature in the `Post` model. Easy. But now, users also want to like comments. You have two choices:

- Copy the like-related features into the `Comment` model
- You implement a generic trait that can be used in any model

Of course, we want the second option. It looks something like that:

```
trait Likeable
{
    public function like(): Like
    {
        // ...
    }

    public function dislike(): void
    {
        // ...
    }

    public function likes(): MorphMany
    {
        // ...
    }
}
```

```
public function likeCount(): int
{
    return $this->likes()->count();
}
}
```

```
class Post extends Model
{
    use Likeable;
}
```

```
class Comment extends Model
{
    use Likeable;
}
```

Now let's say we need to add a chat to the app, and of course, users want to like messages. So we do this:

```
class ChatMessage extends Model
{
    use Likeable;
}
```

This is pretty standard, right? But think about what happened here. We just added new functionality to multiple classes without changing them! We **extended** our classes instead of modifying them. And this is a huge win in the long term. This is why traits and polymorphism in general are amazing tools.

Let's look at another example that uses polymorphism and interfaces. Let's say we're working on an app similar to DoorDash. It has different products and variations. For example, users can order a small batch or large batch of chicken soup. Both have different prices. They can also order pizza with different toppings that modify the price. But they can also order a pretty standard food such as a Cheeseburger without any modification.

Here's an oversimplified database structure:

products:

id	name	price	price_type
1	Chicken soup	7	has_batches
2	Margherita pizza	15	has_toppings
3	Cheeseburger	12	standard

product_batches: this table contains the price modification for different batch sizes

id	product_id	name	price
1	1	Large	2

So a small batch of chicken soup costs \$7 but a large one costs \$12 because of the `product_batches.price` column.

When customers order their food we need to create an `Order` and `OrderItems` for these products:

`orders` : this table is not that important for our purpose so we leave it pretty simple

id	total_price	created_at
1	38	2023-01-08 14:42

`order_items`

id	order_id	product_id	product_batch_id	price
1	1	1	1	9
2	1	2		17
3	1	3		12

toppings : this table contains every toppings users can add to their food:

id	name	price
1	Cheese	1
2	Mushroom	1

order_item_topping : this is a pivot table that associates products with toppings:

id	order_item_id	topping_id
1	2	1
2	2	2

So we can determine the prices using different calculations:

- Chicken soup: `products.price + product_batches.extra_price`
- Margherita pizza: `products.price + sum of the toppings' prices based on the order_item_topping table`
- Cheeseburger: `products.price`

Of course, this is just a hypothetical and oversimplified example, but the technical details are not that important for now so it works for our purpose.

Now let's see how we can calculate prices:

```
class PriceCalculatorService
{
    public function calculatePrice(Order $order): float
    {
        return $order->items()
            ->reduce((float $sum, OrderItem $item) {
                switch ($item->product->type) {
                    case 'standard':
                        return $item->product->price;

                    case 'has_batches':
                        return $item->product->price +
                            $item->product_batch->price;

                    case 'has_toppings':
                        $toppingsSum = $item->toppings
                            ->reduce(function ($sum, Topping $topping) {
                                return $sum + $topping->price;
                            }, 0);

                        return $item->product->price + $toppingsSum;
                }
            }, 0);
    }
}
```

It's not so bad but it has two essential flaws:

- If you're building a DoorDash-like app, just imagine how many times you need to repeat this `switch` statement. If there are other things that depend on the product type (and there are a dozen of them!) it gets even worse.
- What happens when you need to handle a new type of product? Then you need to modify all of those `switch` statements. And this is the bare minimum you need to do, usually, it's a much bigger pain in a project that doesn't rely on polymorphism and the open-closed principle.

Or put it in other words: this architecture violates the open-closed principle. It's absolutely not extensible but requires changing existing (and probably nasty) classes every time a new requirement comes in.

So let's refactor it using OCP and polymorphism. First, we need a class hierarchy that represents the different price types:

```
abstract class PriceType
{
    public function __construct(
        protected readonly OrderItem $orderItem
    ) {}

    abstract public function calculatePrice(): float;
}

class StandardPriceType extends PriceType
{
    public function calculatePrice(): float
    {
        return $this->orderItem->product->price;
    }
}
```

```

class HasBatchesPriceType extends PriceType
{
    public function calculatePrice(): float
    {
        return $this->orderItem->product->price +
            $this->orderItem->product_batch->price;
    }
}

class HasToppingsPriceType extends PriceType
{
    public function calculatePrice(): float
    {
        $toppingsSum = $this->orderItem->toppings
            ->reduce(function (float $sum, Topping $topping) {
                return $sum + $topping->price;
            }, 0);

        return $this->orderItem->product->price + $toppingsSum;
    }
}

```

These classes can calculate the price of a single `OrderItem` that has a `Product`. We need a way to create these classes easily. This is where the factory "design pattern" can be useful:


```
class PriceTypeFactory
{
    public function create(OrderItem $orderItem): PriceType
    {
        switch ($orderItem->product->type)
        {
            case 'standard':
                return new StandardPriceType($orderItem->product);

            case 'has_batches':
                return new HasBatchesPriceType($orderItem->product);

            case 'has_toppings':
                return new HasToppingsPriceType($orderItem->product);
        }
    }
}
```

And now we need a way to create these classes in a model. An attribute accessor is an excellent choice to do so:

```
class OrderItem extends Model
{
    public function priceType(): Attribute
    {
        return new Attribute(
            get: fn () => (new PriceTypeFactory())
                →create($this),
        );
    }
}
```

And finally, we can rewrite the `PriceCalculator` class:

```
class PriceCalculatorService
{
    public function calculatePrice(Order $order): float
    {
        return $order→items
            →reduce(function (float $sum, OrderItem $item) {
                return $sum + $item→price_type→calculatePrice();
            }, 0);
    }
}
```

Do you see what we did? We just eliminated every `switch` statement from the entire application and switched them to separate classes and a simple factory. Now, what happens when a new product type comes in?

- We need to add a new class that extends the abstract `PriceType` class
- We need to add a new case to the `PriceTypeFactory` class

Or put in other words: instead of changing everything, we can **extend** our existing classes with the new functionality

All we needed was a factory, some strategy classes, and a little bit of polymorphism. Of course, now we're only talking about prices that depend on the product type, but usually, there are other things as well. All we need to do is "repeat" this process and introduce another class hierarchy.

Oh and of course, we are cool kids, so let's modernize the factory:

```
class PriceTypeFactory
{
    public function create(OrderItem $item): PriceType
    {
        return match ($item->product->type) {
            'standard' => new StandardPriceType($item->product),
            'has_batches' => new HasBatchesPriceType($item->product),
            'has_toppings' => new HasToppingsPriceType($item->product),
        };
    }
}
```

Or even better we can get rid of the whole thing with the magic strings and we can use an enum that can behave like a factory:

```
enum PriceTypes: string
{
    case Standard = 'standard';
    case HasBatches = 'has_batches';
    case HasToppings = 'has_toppings';

    public function create(OrderItem $item): PriceType
    {
        return match ($this) {
            self::Standard => new StandardPriceType($item),
            self::HasBatches => new HasBatchesPriceType($item),
            self::HasToppings => new HasToppingsPriceType($item),
        };
    }
}
```

The attribute accessor looks like this:

```
class OrderItem extends Model
{
    public function priceType(): Attribute
    {
        return new Attribute(
            get: fn () => PriceTypes::from(
                $this->product->price_type
            )->create($this),
        );
    }
}
```

Liskov Substitution Principle

Each base class can be replaced by its subclasses

It sounds obvious and I think this is the easiest principle to comply with. However, there are some important things.

The principle says that if you have a base class and some subclasses, you should be able to replace the base class with the subclasses anywhere inside your application without any problem.

Consider this scenario:

```
abstract class EmailProvider
{
    abstract public function addSubscriber(User $user): array;

    /**
     * @throws Exception
     */
    abstract public function sendEmail(User $user): void;
}

class MailChimp extends EmailProvider
{
    public function addSubscriber(User $user): array
    {
        // Using MailChimp API
    }
}
```

```
public function sendEmail(User $user): void
{
    // Using MailChimp API
}

class ConvertKit extends EmailProvider
{
    public function addSubscriber(User $user): array
    {
        // Using ConvertKit API
    }

    public function sendEmail(User $user): void
    {
        // Using ConvertKit API
    }
}
```

We have an abstract `EmailProvider` and we use both `MailChimp` and `ConvertKit` for some reason. These classes should behave **exactly** the same way, no matter what.

So if I have a controller that adds a new subscriber:

```

class AuthController
{
    public function register(
        RegisterRequest $request,
        EmailProvider $emailProvider
    ) {
        $user = User::create($request->validated());

        $subscriber = $emailProvider->addSubscriber($user);
    }
}

```

I should be able to use any of these classes without any problem. It should not matter if the current `EmailProvider` is `MailChimp` or `ConvertKit`. I also should be able to switch the argument:

```

public function register(
    RegisterRequest $request,
    ConvertKit $emailProvider
) {}

```

This sounds obvious, however, there are some important thing that needs to be satisfied:

- Same method signatures. In PHP we're not forced to use types so it can happen that the `addSubscriber` method has different types in `MailChimp` compared to `ConvertKit`.
- It's also true for return types. Of course, we can type-hint these, but what about an `array` or a `Collection`? It's not guaranteed that an array contains the same types in multiple classes, right? As you can see, the `addSubscriber` method returns an `array` that contains the subscriber's data received from the APIs. Both `MailChimp` and `ConvertKit` return a different shape. They are arrays, yes, but they are completely

different data structures. So I cannot be 100% sure that `RegisterController` works correctly with any email provider implementation. This is why it's a good idea to have DTOs when working with 3rd parties.

- The same exceptions should be thrown from each method. Since exceptions cannot be type-hinted in the signature it's also a source of difference between these classes.

As you can see the principle is quite simple but it's easy to make mistakes.

Interface Segregation Principle

You should have many small interfaces instead of a few huge ones

The original principle sounds like this: *no code should be forced to depend on methods it does not use* but the practical implication is the definition I gave you. To be honest, this is the easiest principle to follow. In the DashDoor example (see in the open-closed principle chapter), products have a type, such as:

- Standard
- Has batches
- Has toppings
- etc

We had a separate class to handle price calculations for these types. In a real-world application price is not the only thing that depends on the type. There are other things such as:

- Reports
- Data representation
- Inventory management
- Tax and VAT calculations
- Information on the receipt

In the original example, we had these classes:

- `ProductPriceType`
- `StandardProductPrice`
- `HasBatchesProductPrice`
- `HasToppingsProductPrice`

Each of them handles the price calculation for a certain type of product. Imagine if we write some generic `ProductType` class, such as:

- `ProductType`
- `StandardProduct`

- HasBatchesProduct
- HasToppingsProductPrice

And we try to handle **everything** in these classes. So they have functions like this:

```
interface ProductType
{
    public function calculatePrice(Product $product): float;

    public function decreaseInventory(Product $product): void;

    public function calculateTaxes(Product $product): TaxData;

    // ...
}
```

I guess you can see what's the problem. This interface is too big. It handles too many things. Things that are independent of one another. So instead of writing one huge interface to handle everything we separate these responsibilities into smaller ones:

```
interface ProductPriceType
{
    public function calculatePrice(Product $product): float;
}

interface ProductInventoryHandler
{
    public function decreaseInventory(Product $product): void;
}
```

```
interface ProductTaxType
{
    public function calculateTaxes(Product $product): TaxData;
}
```

Another great example of this is PHP traits and how the framework itself, 1st and 3rd party packages and the community uses them:

```
class Broadcast extends Model implements Sendable
{
    use WithData;
    use HasUser;
    use HasAudience;
    use HasPerformance;
}
```

Each of those traits has a pretty small and well-defined interface and it adds a small chunk of functionality to the class. The same goes for the `Sendable` interface.

Dependency Inversion Principle

Depend upon abstraction, not concretions.

Whenever you have a parent class and one or more subclasses you should use the parent class as a dependency. For example:

```
interface class MarketDataProvider
{
    public function getPrice(string $ticker): float;
}

class IexCloud extends MarketDataProvider
{
    public function getPrice(string $ticker): float
    {
        // Using IEX API
    }
}

class Finnhub extends MarketDataProvider
{
    public function getPrice(string $ticker): float
    {
        // Using Finnhub API
    }
}
```

It should be quite straightforward at this point, that we want to do something like this:

```

class CompanyController
{
    public function show(
        Company $company,
        MarketDataProvider $marketDataProvider
    ) {
        $price = $marketDataProvider→getPrice();

        return view('company.show', compact('company', 'price'));
    }
}

```

So every class should depend on the abstract `MarketDataProvider` not on the concrete implementation.

In my opinion, it's important to have these abstractions **even if you have only one implementation** when it comes to 3rd party providers. The reason is that these services and providers change, and you never know what will happen in the future. Just to name a few examples:

- I was using IEX cloud in a financial app a long time ago. I thought that IEX API is the only constant thing in that project. It was great, it was stable, etc. Until they switched from a monthly subscription to usage-based pricing (if I remember correctly). They essentially 3x our expenses. In a project that did not yet have income. So we switched to Finnhub. But of course, we didn't have the correct abstractions so it was a pain in the ass.
- I've been using Gumroad since I began to publish content. I thought I'd never use any other platform. Until this book. This is powered by Paddle. Gumroad 2x'd their prices and Paddle 100% better from an accounting point-of-view.
- I always thought Stripe was the best payment provider in the entire universe. Until I tried Paddle. Now they are my go-to solution and I'm using them in multiple projects.
- A long time ago MailChimp was the industry-standard mail service provider for me. Now I'm using ConvertKit almost exclusively.

- We used Azure at my current workplace until we got free credits from Google that will cover our expenses for the next two years.

Services and providers change and you should be able to handle these changes with minimum effort. Minimum effort means an abstraction above the concrete classes that can be switched without modifying your code.

Building a Dashboard

In this chapter, I'd like to discuss the design and development of a dashboard that shows you sales and revenue information about products. In fact, this is a personal "product" of mine. Earlier, I published a book and it has three different packages. Each package has a different price. I use Gumroad as my e-commerce provider, but handling different packages is not that optimal, so I created a separate Gumroad product for each one. This is not the best solution either because I have three separate products with separate statistics and reports. This is why I developed a small dashboard that shows me:

- How many copies I sold overall
- What was the overall revenue
- A breakdown of the three packages:
 - Copies sold
 - Total revenue
 - Revenue %

If you take a look at this Figma design, you'll have a better idea:



DASHBOARD

SALES

DISCOUNTS



UNITS SOLD

303

TOTAL REVENUE

\$17,307

AVG PRICE

\$57.7

BASIC

12%

\$2,088

TOTAL REVENUE

75

UNITS SOLD

PLUS

43%

\$7,497

TOTAL REVENUE

153

UNITS SOLD

PREMIUM






45%

\$7,722

TOTAL REVENUE

78

UNITS SOLD

CLIENT	DATE	PACKAGE	AMOUNT	DISCOUNT CODE
 test@example.net John Doe	Apr. 16, 2022	Premium	\$99	
 test@example.net John Doe	Apr. 15, 2022	Plus	\$49	
 test@example.net John Doe	Apr. 14, 2022	Basic	\$25	blackfriday2022
 test@example.net John Doe	Apr. 15, 2022	Plus	\$39	blackfriday2022
 test@example.net John Doe	Apr. 14, 2022	Basic	\$29	

This whole page is about one product and "Basic," "Plus," and "Premium" are the packages. So let's design and implement this dashboard!

Data modeling

When integrating with 3rd party data providers (such as Gumroad in this case), you have at least three different solutions about how to get and display the data:

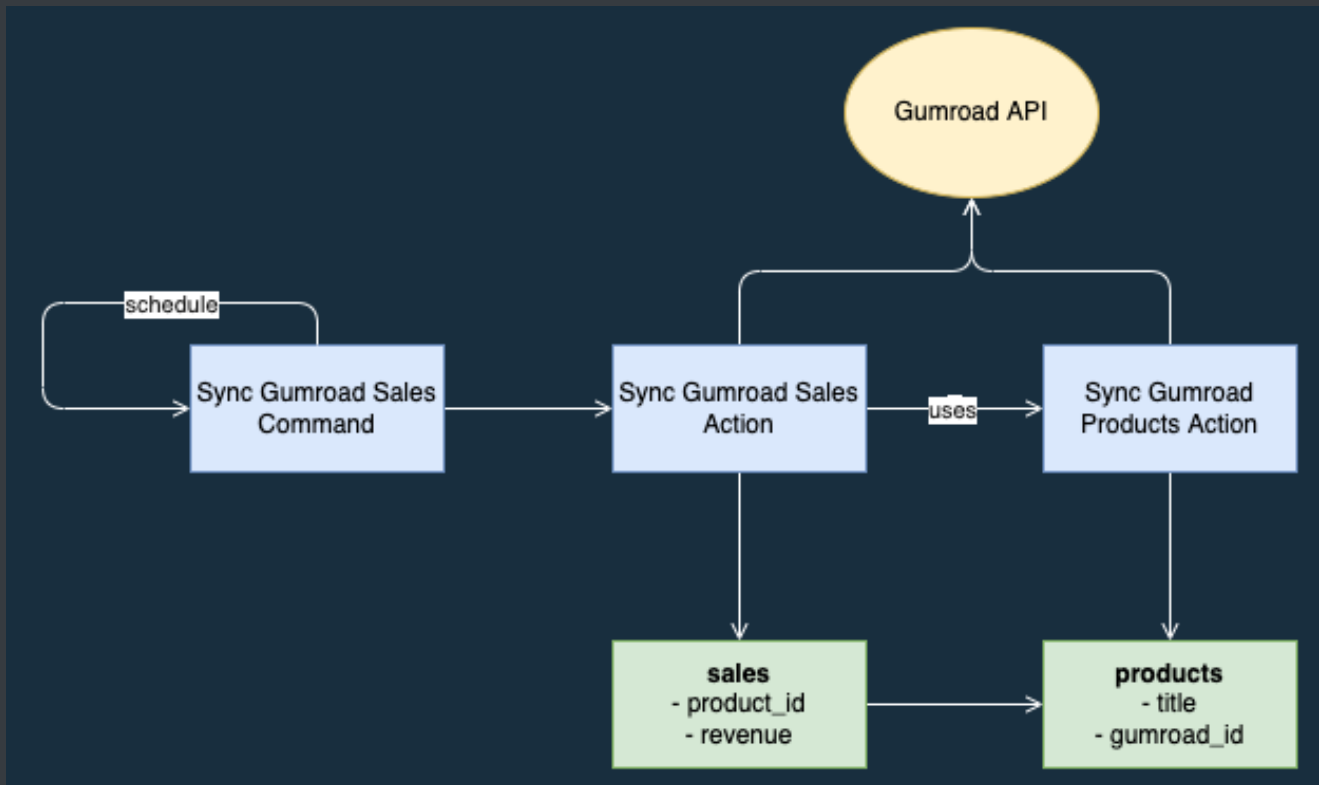
- Send API requests on the fly. So anytime someone opens this dashboard the app will send HTTP requests to Gumroad, then aggregates and displays the important information. This solution is fairly poor from a performance point of view, so it's not a common use case.
- Store 3rd party data in your own database. Instead of on-the-fly API requests you can create your own database and store the important data from Gumroad. This can be implemented in two different ways:
 - Scheduled. In this scenario, you can write a scheduled job that runs every day, hour, or minute (depending on the use case) and sync data from Gumroad to your local database.
 - Real-time. Most e-commerce (or other bigger 3rd parties) offer push (or ping) services. This means whenever a new sale happens they "push" the data to you by calling your API endpoint or using WebSocket.

What you choose depends heavily on the use case, for example:

- If you have a dashboard that is used to gather monthly business information you don't need real-time data.
- On the other hand, if your dashboard is used by a marketing team on the weekend of Black Friday to get sales data and adapt their strategy quickly if needed, then you might need a real-time solution.

In this discussion, we're going with a scheduled solution. However, it'll be very easy to refactor it to a push model, for example. The main difference is that the trigger of the main action is not a scheduled command but an API endpoint.

To implement this dashboard and integration we need the following components:



This is what the whole flow looks like:

- The `SyncGumroadSalesCommand` runs every X minutes.
- It calls the `SyncGumroadSalesAction` class that does two things:
 - It calls the `SyncGumroadProductsAction` which will query all the products from Gumroad and save it into the database.
 - It queries the sales from Gumroad and saves them into the database.

As you can see, we only need two tables, `sales` and `products` .

Gumroad Service

Let's start with integrating the external API. When working with 3rd party services it's a good idea to store the access token and the base URL as an environment variable and a config. In Laravel we have the `services.php` config which can be used for this purpose:

```
return [  
    'gumroad' => [  
        'uri' => env('GUMROAD_URI'),  
        'access_token' => env('GUMROAD_ACCESS_TOKEN'),  
    ],  
];
```

Now we can create the `GumroadService` class that accepts these values in the constructor:

```
namespace App\Services\Gumroad;  
  
class GumroadService  
{  
    public function __construct(  
        private readonly string $accessToken,  
        private readonly string $uri,  
    ) {}  
}
```

But now every time you want to instantiate this class you need to provide these values:

```
new GumroadService('my_token', 'https://api.gumroad.com/v2');
```

Of course, this is not what we want. Fortunately, the Laravel service container makes this very easy:

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        $this->app->when(GumroadService::class)
            ->needs('$accessToken')
            ->give(config('services.gumroad.access_token'));

        $this->app->when(GumroadService::class)
            ->needs('$uri')
            ->give(config('services.gumroad.uri'));
    }
}
```

Now, instead of manually taking care of these values, we can instantiate them like this:

```
$gumroad = app(GumroadService::class);

// Or injecting it into a method
public function index(
    Request $request,
    GumroadService $gumroad
) {
    // ...
}
```

Get Products

The first API we need to use is the `GET /products` :

```
/**
 * @return Collection<ProductData
 */
public function products(): Collection
{
    $products = Http::get("{ $this->uri }/products", [
        'access_token' => $this->accessToken,
    ]->json('products'));

    return collect($products)
        ->map(fn (array $data) => ProductData::fromArray($data));
}
```

As you can see, this method returns a collection of `ProductData` classes. This is a DTO or `DataTransferObject` which is a class that holds data. There's nothing special about DTOs, they just help us eliminate the big, unstructured, and random associative arrays from our code. For example, the `ProductData` is really straightforward since we only need two attributes:

```
namespace App\Services\Gumroad\DataTransferObjects;

class ProductData
{
    public function __construct(
        public readonly string $id,
        public readonly string $name,
    ) {}

    public static function fromArray(array $data): self
    {
        return new self(
            id: $data['id'],
            name: $data['name'],
        );
    }
}
```

By using a DTO such as this one, the only place where we interact with the array given by Gumroad is the `fromArray` method. In every other place, we'll have a nice type-hinted object with auto-completion.

Get Sales

The next API we need to interact with is the `GET /sales`. This is a bit trickier for two reasons:

- It's paginated. So every time you call the API you only get 20 sales and a URL to the next page.
- It requires a `product_id` parameter, so the method in `GumroadService` will accept a collection of products.

```
/**
 * @param Collection<Product> $products
 * @param Carbon|null $after
 * @return Collection<SaleData>
 */
public function sales(
    Collection $products,
    ?Carbon $after = null
): Collection {
    foreach ($products as $product) {
        for ($page = 1; ; $page++) {
            $requestData = [
                'access_token' => $this->accessToken,
                'product_id' => $product->gumroad_id,
                'page' => $page
            ];

            if ($after) {
                $requestData['after'] = $after->format('Y-m-d');
            }
        }
    }
}
```

```

        // ...
    }
}
}

```

A few important things:

- It goes through every product.
- It starts an infinite loop using the `$page` variable.
- It uses the `$page` variable inside the `$requestData` array which is going to be used in the HTTP request later.
- It also accepts an optional `$after` variable, which is very important. Let's say you have sales data from the last 6 months. You already synced everything except the last 4 hours, so you obviously don't want to sync everything every time. To avoid this Gumroad accepts an `after` parameter in the request body. We discuss it in more detail later.

The next part of the method:

```

$response = Http::get("{ $this->uri }/sales", $requestData);

if (!$response->json('next_page_url')) {
    break;
}

```

We actually send the request, and if the `next_page_url` is null it means that there are no more data so we can exit the infinite loop.

Since we need to deal with pagination and we have nested loops, we need to keep track of the sales for a given product:


```
foreach ($products as $product) {  
    $salesByProduct = []; // ← Important  
  
    for ($page = 1; ; $page++) {  
        // ...  
  
        $response = Http::get("{${this->uri}"/sales", $requestData);  
  
        // Important  
        $salesByProduct = [  
            ... $salesByProduct, // Sales from previous pages  
            ... $response->json('sales'), // Sales from the current  
page  
        ];  
    }  
}
```

So I introduced the `$salesByProduct` array which holds every sale made by a particular product. After I send the request I merge the new sales with the old ones. New means the current page, and old means the previous pages. So the `$salesByProduct` array contains every sale made by one product. As you can guess, we need every sale made by every product.

```

$sales = []; // ← Important

foreach ($products as $product) {
    $salesByProduct = [];

    for ($page = 1; ; $page++) {
        // ...

        $salesByProduct = [
            ... $salesByProduct,
            ... $response->json('sales'),
        ];
    }

    // Important
    $sales = [
        ... $sales, // Sales by previous products
        ... $salesByProduct, // Sales by current product
    ];
}

return collect($sales)->sortByDesc('date');

```

So I essentially did the same thing with the `$sales` array as earlier with the `$salesByProduct`.

The last thing we need to do is introduce a new DTO called `SaleData` so we can return a collection of DTOs from this method as well:

```

namespace App\Services\Gumroad\DataTransferObjects;

class SaleData
{
    public function __construct(
        public readonly string $email,
        public readonly ?string $full_name,
        public readonly float $revenue,
        public readonly Carbon $date,
        public readonly ProductData $product,
    ) {}

    public static function make(
        array $data,
        Product $product
    ): self {
        return new self(
            email: $data['email'],
            full_name: Arr::get($data, 'full_name'),
            revenue: $data['price'] / 100,
            date: Carbon::parse($data['created_at']),
            product: ProductData::fromModel($product),
        );
    }
}

```

Since every sale has a product associated with it, we need to accept a `Product` instance. As you can see, this DTO has a nested `ProductData` property. For this reason, the `make` function accepts a `$data` array and the `Product`. This is why it's called `make` instead of `fromArray`. This is a rule of thumb for me. DTOs usually have three kinds of factory

methods:

- `fromArray(array $data)` We have seen this earlier in the `ProductData` class. This is the most common one.
- `fromModel(Order $order)` This one creates a DTO from a model instance. In this example, it creates an `OrderData` from an `Order` model.
- `make(...)` This one always accepts more than one argument.

Now that we have the `SaleData` class, this is what the `sales` method looks like:

```
/**
 * @param Collection<Product> $products
 * @param Carbon|null $after
 * @return Collection<SaleData>
 */
public function sales(
    Collection $products,
    ?Carbon $after = null
): Collection {
    $sales = [];

    foreach ($products as $product) {
        $salesByProduct = [];

        for ($page = 1; ; $page++) {
            $requestData = [
                'access_token' => $this->accessToken,
                'product_id' => $product->gumroad_id,
                'page' => $page
            ];
```

```

if ($after) {
    $requestData['after'] = $after->format('Y-m-d');
}

$response = Http::get(
    "{$this->uri}/sales",
    $requestData
);

$salesByProduct = [
    ... $salesByProduct,
    ... $response->json('sales'),
];

if (!$response->json('next_page_url')) {
    break;
}
}

$dtos = collect($salesByProduct)
    ->map(fn (array $sale) =>
        SaleData::make($sale, $product)
    );

$sales = [
    ... $sales,
    ... $dtos,
];
}

```

```
return collect($sales)→sortByDesc('date');
}
```

It's a bit long and the body of the loop can clearly be a separate function. It makes sense, in my opinion, to introduce a new method called `salesByProduct()`. So this is the final form of our method:

```
/**
 * @param Collection<Product> $products
 * @param Carbon|null $after
 * @return Collection<SaleData>
 */
public function sales(
    Collection $products,
    ?Carbon $after = null
): Collection {
    $sales = [];

    foreach ($products as $product) {
        $sales = [
            ... $sales,
            ... $this→salesByProduct($product, $after),
        ];
    }

    return collect($sales)→sortByDesc('date');
}
```

Basically, I just moved the loop's body into a new function. Now that we have the `GumroadService` we can implement the sync actions.

Sync With Gumroad

Sync Gumroad Products

Syncing products is really straightforward:

```
namespace App\Actions;

class SyncGumroadProductsAction
{
    public function __construct(
        private readonly GumroadService $gumroad
    ) {}

    public function execute(): void
    {
        foreach ($this->gumroad->products() as $product) {
            Product::updateOrCreate(
                [
                    'gumroad_id' => $product->id,
                ],
                [
                    'gumroad_id' => $product->id,
                    'title' => $product->name,
                ]
            );
        }
    }
}
```

It couldn't be easier. It requests the products and then inserts or updates them in the database. If you don't know `updateOrCreate`, here's how it works:

- If it finds a row with the attributes given in the first array (`gumroad_id` in this case) it's going to perform an update query. If there's no result, it's going to perform an insert query.
- The update or insert will use the second array.

Sync Gumroad Sales

As I said earlier, the `sales` method in the `GumroadService` class accepts an optional `$after` parameter. Since the sync command will run on a scheduled basis we only need to query sales made in the last X hours. Or in other words: **sales after the last sync**. This is why I introduced a new table, the `gumroad_syncs`. We will create a new row in this table after a sync has been successfully finished. The table looks like this:

id	synced_at
1	2022-05-12 13:11:32
2	2022-05-13 14:54:01
3	2022-05-14 21:09:12

Of course, you don't need to store every occurrence, you can have only one row that stores the last sync. However, having a detailed history sometimes makes the debugging process easier.

That being said, here's what the `SyncGumroadSalesAction` looks like:


```

        ],
        [
            'product_id' => $product->id,
            'customer_email' => $sale->email,
            'customer_name' => $sale->full_name,
            'revenue' => $sale->revenue,
            'sold_at' => $sale->date,
        ]
    );
}

GumroadSync::create([
    'synced_at' => Sale::latest('sold_at')->first()->sold_at,
]);
}
}

```

Step-by-step:

- Every time this action is triggered it will sync the products first.
- It requests the sales from Gumroad API using the last sync date.
- For each sale, it creates a new record in the `sales` table.
- It creates a new record into the `gumroad_syncs` table.

The last piece of the puzzle is the actual command that runs every X hours:

```

namespace App\Console\Commands;

class SyncGumroadSalesCommand extends Command
{
    protected $signature = 'sync';
    protected $description = 'Sync sales and products from
Gumroad';

    public function handle(
        SyncGumroadSalesAction $syncGumroadSales
    ) {
        $syncGumroadSales->execute();

        return self::SUCCESS;
    }
}

```

As you can see, it only calls the action. This command can be used from the terminal by `php artisan sync` and it can be scheduled in the `Console/Kernel` :

```

namespace App\Console;

class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule)
    {
        $schedule->command('sync')->hourly();
    }
}

```

Dashboard

Now we have every data needed to show the dashboard. If you take a look at the design you see that it shows you more than one product. This means we need to accept product IDs in the URL. This is the request:

```
namespace App\Http\Requests;

class GetDashboardRequest extends FormRequest
{
    /**
     * @return Collection<Product>
     */
    public function products(): Collection
    {
        return Product::whereIn('id', $this->product_ids)->get();
    }

    public function rules()
    {
        return [
            'product_ids' => ['required', 'array'],
            'product_ids.*' => ['exists:products,id'],
        ];
    }
}
```

I often write simple getters such as the `products` method. Some other examples:

So the main idea here:

- Converting collections from arrays.
- Dealing with null values.
- Converting strings to Carbon.
- Querying models. This means **only simple where id = X queries**. Nothing more complicated!

So the important thing is: **no business logic in requests**. These getters and helpers will make your controllers a little bit cleaner. In our example, there's no need to query products by ids, since it's already done in the request.

Back to our dashboard. Now we have a request, so let's use it in the controller:

```
namespace App\Http\Controllers;

use App\Http\Requests\GetDashboardRequest;
use App\ViewModels\GetDashboardViewModel;

class GetDashboardController extends Controller
{
    public function __invoke(GetDashboardRequest $request)
    {
        $viewModel = new GetDashboardViewModel(
            $request->products()
        );

        return view('dashboard', [
            'model' => $viewModel->toArray(),
        ]);
    }
}
```

It's a very small application, but I almost always use view models by default. What is a view model, you might ask? `ViewModel` is a very clever way to handle view-related data. I'm not talking about Blade views exclusively. You can think of a view model as a data container responding to a specific request such as the 'Get Dashboard' request. They can be used in both SPAs (Inertia.js included) and full-stack MVC applications.

Let's think about what data we need on the dashboard:

- Sales summary. This is the header with the following data:
 - Units sold
 - Total Revenue
 - Average Price
- Products summary. The three cards that show:
 - The product's name
 - Total revenue
 - Units sold
 - Revenue contribution in percentage
- List of the last 10 sales

This is our UI and we can express this in the view model by creating methods such as:

- `salesSummary`
- `productsSummary`
- `sales`

For example:

```

class GetDashboardViewModel extends ViewModel
{
    /**
     * @var Collection<Sale>
     */
    private Collection $sales;

    public function __construct(
        private readonly Collection $products
    ) {}

    public function salesSummary(): SalesSummaryData
    {
        $sales = Sale::latest('sold_at')
            →whereIn('product_id', $this→products→pluck('id'))
            →get();

        $totalRevenue = $sales→sum('revenue');

        $unitsSold = $sales→count();

        return new SalesSummaryData(
            units_sold: $unitsSold,
            total_revenue: round($totalRevenue, 0),
            average_price: round($totalRevenue / $unitsSold, 1),
        );
    }
}

```

The constructor accepts a collection of products. They come from the request discussed

earlier. The `salesSummary` method performs a query and runs some calculations. After that, it returns a simple DTO:

```
namespace App\DataTransferObject;

class SalesSummaryData
{
    public function __construct(
        public readonly int $units_sold,
        public readonly float $total_revenue,
        public readonly float $average_price,
    ) {}
}
```

As you can see, this class extends an abstract `ViewModel` class. This only provides a `toArray` method which converts every public method into an array key. So by calling the `toArray` on this class we get something like that:

```
[
    'sales' => [
        'units_sold' => 10,
        'total_revenue' => 200,
        'average_price' => 20,
    ],
];
```

Later, I'll show you how it's done, but now you can understand why the `toArray` is called in the controller:

```
class GetDashboardController extends Controller
{
    public function __invoke(GetDashboardRequest $request)
    {
        $viewModel = new GetDashboardViewModel(
            $request->products()
        );

        return view('dashboard', [
            'model' => $viewModel->toArray(),
        ]);
    }
}
```

In this example, I use Blade so the data will be passed as an array. In the case of a JSON API, Laravel will convert arrays into JSON, so this method works in both cases.

Now, let's see what else is in the view model. First of all, I need the sales in multiple places and I don't want to query them multiple times, so I put the query in the constructor:

```

class GetDashboardViewModel extends ViewModel
{
    /**
     * @var Collection<Sale>
     */
    private Collection $sales;

    public function __construct(
        private readonly Collection $products
    ) {
        $this->sales = Sale::latest('sold_at')
            ->whereIn('product_id', $this->products->pluck('id'))
            ->get();
    }
}

```

It's private property and it contains every sale. Private properties won't be included in the resulting array, so it's just private data that can be used in this class. I also need the total revenue in multiple places (in the sales summary, and in the product summary as well), so I created a private method:

```

private function totalRevenue(): float
{
    return $this->sales->sum('revenue');
}

```

With these two functions we can refactor the `salesSummary` :

```

public function salesSummary(): SalesSummaryData
{
    $totalRevenue = $this->totalRevenue();

    $unitsSold = $this->sales->count();

    return new SalesSummaryData(
        units_sold: $unitsSold,
        total_revenue: round($totalRevenue, 0),
        average_price: round($totalRevenue / $unitsSold, 1),
    );
}

```

We can also write the function that returns the 10 most recent sales:

```

/**
 * @return Collection<SaleData>
 */
public function sales(): Collection
{
    return $this->sales
        ->take(10)
        ->map(fn (Sale $sale) => SaleData::fromModel($sale));
}

```

It takes the last 10 sales and maps them into DTOs. The last method is the `productSummaries` :

```

/**
 * @return Collection<ProductSaleSummaryData>
 */
public function productSummaries(): Collection
{
    $totalRevenue = $this->totalRevenue();

    return $this->products
        ->map(function (Product $product) use ($totalRevenue) {
            $productRevenue = $product->sales->sum('revenue');

            return new ProductSaleSummaryData(
                product: ProductData::fromModel($product),
                total_revenue: $productRevenue,
                units_sold: $product->sales->count(),
                total_revenue_contribution: $productRevenue /
                    $totalRevenue,
            );
        });
}

```

It creates a `ProductSaleSummaryData` DTO for each product. The DTO is pretty simple:

```
namespace App\DataTransferObject;

class ProductSaleSummaryData
{
    public function __construct(
        public readonly ProductData $product,
        public readonly float $total_revenue,
        public readonly int $units_sold,
        public readonly float $total_revenue_contribution,
    ) {}
}
```

ViewModels can help your project in two ways:

- Your code is one step closer to the domain language. So when a product manager says "on the dashboard page," you immediately know that they talk about the `GetDashboardViewModel` .
- It can be an excellent addition to have the exact structure of your UI expressed as classes.

As I promised earlier, here's the parent `ViewModel` class:

```

abstract class ViewModel implements Arrayable
{
    public function toArray(): array
    {
        return collect((new ReflectionClass($this))→getMethods())
            →reject(fn (ReflectionMethod $method) ⇒
                in_array(
                    $method→getName(),
                    ['__construct', '__invoke', 'toArray']
                )
            )
            →filter(fn (ReflectionMethod $method) ⇒
                in_array(
                    'public',
                    Reflection::getModifierNames(
                        $method→getModifiers()
                    )
                )
            )
            →mapWithKeys(fn (ReflectionMethod $method) ⇒ [
                Str::snake($method→getName()) ⇒ $this→{$method-
>getName()}()
            ])
            →toArray();
    }
}

```

This method is defined in the parent `ViewModel` class that every view model extends. We only need to implement the `Arrayable` interface, and Laravel will take care of the rest. I don't want to go into too many details about how the reflection API works, but the main logic is this:

- We want every method from the view model except `__construct`, `__invoke`, and `toArray`.
- We also want to reject every `private` or `protected` method. Only `public` methods represent data properties.
- We want the array keys to be in `snake_case`. So a method called `salesSummary` will become `sales_summary` in the array and the view.

This is how the view model gets converted into an array and passed to the Blade (or Vue) view as a property.

Handling Discounts and Coupons

In this chapter, I'd like to discuss a common scenario in e-commerce and enterprise applications: applying discounts. In my experience, this is a feature that most of the time starts simple but over time it gets very complicated. But the worst of it: it often results in monstrous code.

In this article I'll talk about coupons with the following behaviors:

- They have a type. It's either a fixed amount discount or a percentage-based.
- They can have a condition. For example, you can only apply a particular coupon if the order's total amount is bigger than \$99.

These are pretty common use cases in real-world applications, so let's take a look at them!

Data Modeling

We're going to have the following tables:



Coupons can be applied to orders and each order has many order items, where each item contains a product. This is a pretty common table structure when dealing with orders and discounts.

Each coupon has one of the following types:

- `fix_amount`: in this case, the `discount` attribute contains the dollar amount of the

actual discount.

- `percent` : in this case, the `discount` attribute contains a percentage value.

When applying a `fix_amount` coupon to an order the `orders.discounted_price` attribute will be: `orders.total_price - coupons.discount`.

When applying a `percent` coupon to an order the `orders.discounted_price` attribute will be: `orders.total_price * (1 - coupons.discount)`.

Coupons also have one condition associated with them. It's a JSON column in the following format:

```
{
  "type": "order_total_price",
  "value": 99
}

{
  "type": "lifetime_order_amount",
  "value": 990
}
```

These are the two kinds of conditions the sample app will support:

- `order_total_price` : this coupon can only be applied when the current order's `total_price` attribute is greater than the `value` or \$99 in this example.
- `lifetime_order_amount` : this coupon can only be applied when the lifetime order value of the current customer is greater than `value` or \$990 in this case.

You can store these conditions in a separate table, but in this example, it's perfectly fine to store them in the `coupons` table as a column.

Expired Coupons

There are two more rules I didn't mention earlier:

- Each coupon has an expiration date.
- And a state. It's either active or inactive. Right now, the business logic is this: each coupon is for one-time use.

These are pretty easy business rules, but they give us a good opportunity to practice TDD.

So without any existing code, let's write a test that makes sure that an expired coupon cannot be applied. First, we need some setup code, and data to use:

```
namespace Tests\Feature;

class ApplyCouponTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function it_should_not_apply_an_expired_coupon()
    {
        $user = User::factory()->create();
        $token = $user
            ->createToken('test_token', ['*'])
            ->plainTextToken;

        // Given
        $order = Order::factory([
            'total_price' => 100,
            'discounted_price' => 100,
            'discount_value' => 0,
```

```

        'coupon_id' => null,
    ]->create();

    $coupon = Coupon::factory([
        'type' => CouponTypes::Percentage,
        'discount' => 10,
        // This is the important part. The coupon is already
        expired.
        'expires_at' => now()->subDay(),
    ]->create();
    }
}

```

Each test needs a user and a token since they're going to call an API. After I have a token, I create an `Order` and a `Coupon`. The important thing is that this coupon expired yesterday.

After that, everything is ready to call a non-existing API:

```

// When
$this->patchJson(
    route('coupons.apply', compact('order', 'coupon')),
    [],
    [
        'Authorization' => "Bearer {$token}",
    ]
)->assertStatus(Response::HTTP_UNPROCESSABLE_ENTITY);

```

It's going to be a `PATCH` request since it does not create any new resources but modifies an existing `Order`. The `patchJson` method comes from the parent `TestCase` class provided by Laravel. It sends a `PATCH` request in JSON format. The arguments in order:

- The URL we want to call. I use the `route` helper, and I recommend you to do the same. Our URL will look something like this: `/orders/1/coupons/10/apply`. This will apply the coupon with the ID of 10 to the order with the ID of 1.
- The second argument is the body of the request. We don't need it.
- The third one is the headers array. This is where I use the `$token` created earlier.

If I call the API with an expired coupon, I expect a `422 Unprocessable Entity` response. This is done by the `assertStatus` helper method. As a sanity check, I also make sure that the order's `discounted_price` attribute is unchanged:

```
$this->assertDatabaseHas('orders', [  
    'id' => $order->id,  
    'total_price' => 100,  
    'discounted_price' => 100,  
    'discount_value' => 0,  
    'coupon_id' => null,  
]);
```

The `assertDatabaseHas` method will run a `select` query with the attributes. If it finds a record, the test will pass. If it does not find any record, the test will fail.

Applying a coupon means two things:

- The `discounted_price` attribute will change.
- The `coupon_id` is set to the coupon's ID.

In this assertion, we assume none of them happened. To make this test green, we need five things:

- A route
- A controller
- An action
- An if statement
- An exception

The route:

```
Route::middleware('auth:sanctum')->group(function () {
    Route::patch(
        '/orders/{order}/coupons/{coupon}/apply',
        ApplyCouponController::class
    )->name('coupons.apply');
});
```

The controller:

```
namespace App\Http\Controllers;

class ApplyCouponController extends Controller
{
    public function __invoke(
        Order $order,
        Coupon $coupon,
        ApplyCouponAction $applyCoupon
    ) {
        try {
            $applyCoupon->execute($coupon, $order);
        } catch (CannotApplyCouponException) {
            abort(Response::HTTP_UNPROCESSABLE_ENTITY);
        }
    }
}
```

```

    }
}

```

This time, I'm using an invokable controller. Usually, this is how I approach controllers:

- I put CRUD action inside a resource controller.
- I create separate invokable controllers for other, non-crud actions.

The action and the if statement:

```

namespace App\Action;

class ApplyCouponAction
{
    public function execute(Coupon $coupon, Order $order)
    {
        if ($coupon->expires_at->isPast()) {
            throw CannotApplyCouponException::because(
                'Coupon is expired'
            );
        }
    }
}

```

And finally, the exception:

In this approach, every "reason" has a separate factory function. It results in longer exception classes, but it's very easy to find where you use the separate "reasons,"

Inactive Coupons

The next thing we need to check is the status of the coupon. It's either active or inactive and users cannot apply inactive coupons.

The test is very similar to the previous one:

```
/** @test */
public function it_should_not_apply_an_inactive_coupon()
{
    $user = User::factory()>create();
    $token = $user
        >createToken('test_token', ['*'])
        >plainTextToken;

    // Given
    $order = Order::factory([
        'total_price' => 100,
        'discounted_price' => 100,
        'discount_value' => 0,
        'coupon_id' => null,
    ])>create();

    $coupon = Coupon::factory([
        'type' => CouponTypes::Percentage,
        'discount' => 10,
        // This is the important line
        'active' => false,
    ])>create();

    // When
```

```
$this->patchJson(
    route('coupons.apply', compact('order', 'coupon')),
    [],
    [
        'Authorization' => "Bearer {$token}",
    ]
)->assertStatus(Response::HTTP_UNPROCESSABLE_ENTITY);

// Then
$this->assertDatabaseHas('orders', [
    'id' => $order->id,
    'total_price' => 100,
    'discounted_price' => 100,
    'discount_value' => 0,
    'coupon_id' => null,
]);
}
```

These are the same steps as in the previous test:

- Creating an order
- Creating an **inactive** coupon
- Calling the apply API
- Asserting a 422 response code
- Asserting that the order has not changed

Implementing the feature is even easier:

```
namespace App\Action;

class ApplyCouponAction
{
    public function execute(Coupon $coupon, Order $order): Order
    {
        if ($coupon->expires_at->isPast()) {
            throw CannotApplyCouponException::because(
                'Coupon is expired'
            );
        }

        if (!$coupon->active) {
            throw CannotApplyCouponException::because(
                'Coupon is inactive'
            );
        }
    }
}
```

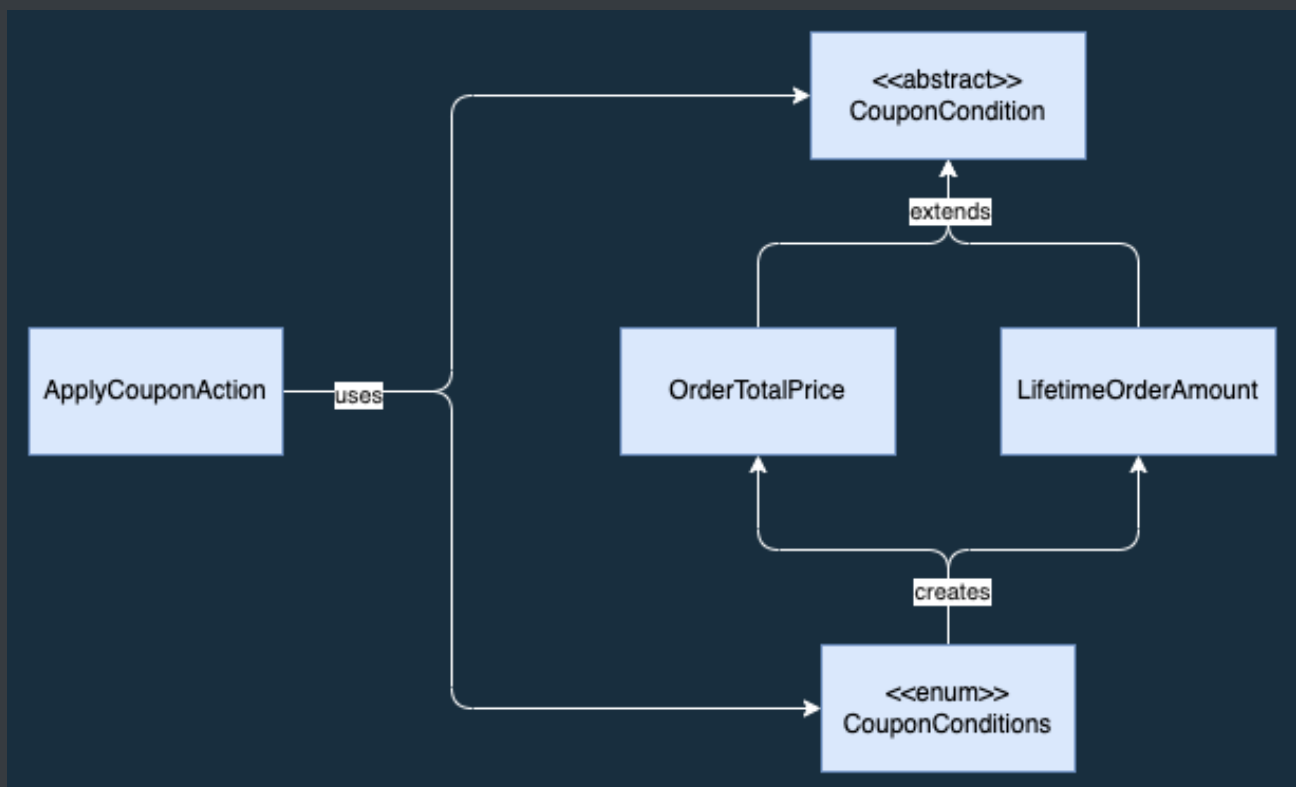
As you can see, it's very easy to test and implement these edge-cases, but they are **crucial**! This is why I usually start with these business rules.

Coupon Conditions

As I said earlier, there are kinds of conditions:

- `order_total_price` : this coupon can only be applied when the current order's `total_price` attribute is greater than the `value` or \$99 in this example.
- `lifetime_order_amount` : this coupon can only be applied when the lifetime order value of the current customer is greater than `value` or \$990 in this case.

I'm going to use an enum and the strategy pattern to handle these scenarios. This is how things are going to work:



All right, I admit it looks a bit complicated, so let's break it down.

CouponCondition strategy

- `CouponCondition` is an abstract base class with one method called `check`. This class describes how you can interact with a condition. You can check if it passes or not given the current order and coupon.
- `OrderTotalPrice` is a concrete implementation of the `CouponCondition`. This class will check if the current order's amount is big enough to apply a particular coupon.

- `LifetimeOrderAmount` is also an implementation of the `CouponCondition`. This one will check if the lifetime order value of the current customer is big enough to apply the given coupon.

CouponConditions enum

This enum has two cases: `order_total_price` and `lifetime_order_amount`. So it represents all of the possible conditions in our application. But more importantly, it can also be used as a factory. So the enum will create instances of the `OrderTotalPrice` and the `LifetimeOrderAmount` classes.

ApplyCouponAction

You've already seen this class. This is where the action happens. It will use the enum to create the strategy then it calls the `check` method. The important thing is that it won't depend on the concrete implementations but only on the abstract `CouponCondition` class. So the action doesn't care about if the current condition is `order_total_price` or `lifetime_order_amount` since both have the same interface. So this action can use any of type conditions if they implement the `CouponCondition` base class. This is called Dependency Inversion Principle and it is the letter D from SOLID.

Before we dive in, I'd like to show another thing related to coupon conditions. If you remember they are stored as JSON in the database:

```
{
  "type": "order_total_price",
  "value": 99
}
```

In situations like this, you should almost always cast them into objects so you don't have to worry about arrays. To achieve this, I created a simple value object:

```

namespace App\ValueObject;

class CouponCondition
{
    public function __construct(
        public readonly CouponConditions $type,
        public readonly float $value,
    ) {}

    public static function fromArray(array $data): self
    {
        return new self(
            type: CouponConditions::from($data['type']),
            value: $data['value'],
        );
    }

    public function toArray(): array
    {
        return get_object_vars($this);
    }
}

```

This object doesn't do anything, it just replaces an unstructured array. But as you can see, the `$type` property is not a string but an enum. This has two advantages:

- Type-hints. You know exactly what the `$type` means in this context.
- Discoverability. You just `Cmd + Click` on the word `CouponConditions` and you see exactly what it is all about.
- Easier to use. Later, in the action, you'll see what I mean.

By the way, what's the difference between value objects and data transfer objects? A value object never has an ID. Two VOs equality is purely based on the value inside them. On the other hand, a DTO should have an ID since most of the time it represents a model.

The next step is to cast the `condition` attribute in the `Coupon` into a `CouponCondition` value object. To achieve this we can use a simple Eloquent cast:

```
namespace App\Models\Casts;

use App\ValueObject\CouponCondition;

class CouponConditionCast implements CastsAttributes
{
    public function get($model, $key, $value, $attributes)
    {
        return CouponCondition::fromArray(
            json_decode($value, true)
        );
    }

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

There are only two methods in every Eloquent cast:

- `get` is called when you access the property in the model. In this method, we're creating a value object from a raw JSON string.
- `set` is called when you set a property in the model. In this method, we're creating a JSON string from a value object.

Now, let's implement the whole workflow!

CouponCondition strategy

As discussed before, the base class only contains a constructor and a `check` method:

```
namespace App\CouponConditions;

abstract class CouponCondition
{
    public function __construct(public readonly Coupon $coupon)
    {
    }

    abstract public function check(Order $order): void;
}
```

Why is the `$coupon` inside the constructor but the `$order` is in the `check` method? Right now, there's no particular reason. This class is small so it doesn't matter that much.

However, it seemed a bit cleaner to me:

- `constructor` : create a coupon condition for this coupon.
- `check` : check this order against the condition.

It makes sense to me. One of the implementations of this abstract class is the `OrderTotalPrice` :

```

namespace App\CouponConditions;

class OrderTotalPrice extends CouponCondition
{
    public function check(Order $order): void
    {
        if ($order->total_price < $this->coupon->condition->value)
        {
            throw CannotApplyCouponException::because(
                "This coupon can only be applied if the order
                amount is greater than
                {$this->coupon->condition->value}"
            );
        }
    }
}

```

It's that easy. We only need to compare the order's total price with the coupon condition's value. Remember, the `$this->coupon->condition` will return a value object that has a `value` property.

The other condition is the `LifetimeOrderAmount` :

```

namespace App\CouponConditions;

use App\Exceptions\CannotApplyCouponException;
use App\Models\Order;

class LifetimeOrderAmount extends CouponCondition
{

```

```

public function check(Order $order): void
{
    $lifetimeOrder = $order
        →user
        →orders
        →sum('total_price');

    if ($lifetimeOrder < $this→coupon→condition→value) {
        throw CannotApplyCouponException::because(
            "This coupon can only be applied if you have already
            ordered at least
            {$this→coupon→condition→value}"
        );
    }
}

```

It's also pretty straightforward. It queries the sum of all orders associated with the user and compares it with the condition's value.

The goal of this article is not to write some complicated business code and queries but to give you a framework on how to structure a solution like this one. But I guess you can already see the advantage of the strategy pattern: each implementation has a dedicated class. It keeps your code so clean that it'll be a joy to work on.

CouponConditions enum

Now that we have the strategies we need a factory that can create them. An enum is a perfect tool for this:

```

namespace App\Enums;

enum CouponConditions: string
{
    case OrderTotalPrice = 'order_total_price';
    case LifetimeOrderAmount = 'lifetime_order_amount';

    public function createCouponCondition(
        Coupon $coupon
    ): CouponCondition {
        return match ($this) {
            self::OrderTotalPrice => new OrderTotalPrice($coupon),
            self::LifetimeOrderAmount => new
LifetimeOrderAmount($coupon),
        };
    }
}

```

It can be used such as:

```

$condition = CouponConditions::from('order_total_price')
    ->createCouponCondition($coupon);

$condition->check();

```

ApplyCouponAction

And finally, we can put everything together in the `ApplyCouponAction` :

```
namespace App\Action;

class ApplyCouponAction
{
    public function execute(Coupon $coupon, Order $order): Order
    {
        if ($coupon->expires_at->isPast()) {
            throw CannotApplyCouponException::because(
                'Coupon is expired'
            );
        }

        if (!$coupon->active) {
            throw CannotApplyCouponException::because(
                'Coupon is inactive'
            );
        }

        $condition = $coupon
            ->condition
            ->type
            ->createCouponCondition($coupon);

        $condition->check($order);
    }
}
```

Let's summarize everything:

- The `Coupon` model uses the `CouponConditionCast` so the `condition` attribute is cast to a `CouponCondition` value object
- The `CouponCondition` value object has a `$type` property and is a `CouponConditions` enum
- The `CouponConditions` enum has a factory function that creates instances of the `CouponCondition` strategy
- The `CouponCondition` strategy has a `check` method that will throw an exception if the given order does not pass the conditions

Now you can clearly see the advantages and disadvantages of this solution.

Advantages:

- Everything is simple.
- The action is probably one of the simplest classes you've ever seen, but it already takes care of four edge-cases:
 - Coupon is expired
 - Coupon is inactive
 - Order's total price is too low
 - Customer's lifetime value is too low
- Everything has its own place and we have a lot of small classes. It's very easy to work with such classes.

Disadvantages:

- Naming things is hard. At this point, we have:
 - `CouponConditionCast`
 - `CouponConditions` enum
 - `CouponCondition` strategy
 - `CouponCondition` value object
- A lot of classes. I mean, I rather work with ten 100-line long classes than a huge one, but it's a bit harder to navigate.

Coupon types

Now that we have handled every important edge-cases we can start actually applying coupons. There are two kinds of coupons:

- Percentage
- Fix amount

Two types of coupons. Or in other words: two implementations of an abstraction. Does it ring any bell to you? Yes! Enums and strategy!

```
namespace App\Enums;

enum CouponTypes: string
{
    case Percentage = 'percentage';
    case FixAmount = 'fix_amount';

    public function createCouponType(Coupon $coupon): CouponType
    {
        return match ($this) {
            self::FixAmount => new FixAmount($coupon),
            self::Percentage => new Percentage($coupon),
        };
    }
}
```

At this point, you can see a pattern. Very often enums have some cases and a factory method. This factory creates a `CouponType` instance:


```

namespace App\CouponType;

use App\ValueObject\Discount;

abstract class CouponType
{
    public function __construct(
        protected readonly Coupon $coupon
    ) {}

    abstract public function getDiscount(Order $order): Discount;
}

```

We'll use the coupon types to calculate the actual discount for a particular order. This class follows the same pattern as earlier:

- constructor : create a coupon type from this `$coupon`
- getDiscount : get the discount for this `$order`

As you can see, the `getDiscount` method returns a value object:

```

namespace App\ValueObject;

class Discount
{
    public function __construct(
        public readonly float $discountedPrice,
        public readonly float $discountValue,
    ) {}
}

```

The `getDiscount` method will return the new, discounted price for the order and the discount value. An example:

- Order total price: \$100
- Discount: 10%
- Discounted price: \$90
- Discount value: \$10

Now let's see the implementation of the percentage discount type:

```
namespace App\CouponType;

class Percentage extends CouponType
{
    public function getDiscount(Order $order): Discount
    {
        $discount = $order->total_price * $this->coupon->discount;

        $discountedPrice = $order->total_price - $discount;

        return new Discount($discountedPrice, $discount);
    }
}
```

It's not that complicated and the other one is also pretty simple:

```
namespace App\CouponType;

class FixAmount extends CouponType
{
    public function getDiscount(Order $order): Discount
    {
        $discount = min(
            $this->coupon->discount,
            $order->total_price
        );

        $discountedPrice = $order->total_price - $discount;

        return new Discount($discountedPrice, $discount);
    }
}
```

However, there's one important thing. It's possible that a coupon has a bigger discount amount than the order's total price itself. To avoid negative results I use the `min` function. It always returns the minimum of the numbers. So if the coupon's discount is the bigger the order's total price will be the discount value.

Before moving on to the action, let's cast the `Coupon` model's `type` attribute to the enum:

```

namespace App\Models;

class Coupon extends Model
{
    use HasFactory;

    protected $casts = [
        'type' => CouponTypes::class,
        'expires_at' => 'datetime',
        'active' => 'bool',
        'condition' => CouponConditionCast::class,
    ];
}

```

Now any time you access the `$coupon->type` you'll get an enum instance instead of a string. These little tricks make the action pretty simple and easy to read:

```

namespace App\Action;

class ApplyCouponAction
{
    public function execute(Coupon $coupon, Order $order): Order
    {
        if ($coupon->expires_at->isPast()) {
            throw CannotApplyCouponException::because(
                'Coupon is expired'
            );
        }
    }
}

```



```
$discount = $coupon->type
    ->createCouponType($coupon)
    ->getDiscount($order);
```

And now we can finally update the order's discount-related attributes:

```
namespace App\Models;

use App\ValueObject\Discount;

class Order extends Model
{
    public function updateDiscount(
        Discount $discount,
        Coupon $coupon
    ): void {
        $this->discounted_price = $discount->discountedPrice;
        $this->discount_value = $discount->discountValue;
        $this->coupon_id = $coupon->id;

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

Here we make use of the `Discount` value object as well, so we don't have to fight with arrays.

Working with 3rd Parties

In this chapter, I'd like to focus on working with 3rd party services. In almost every project we need to integrate our application with some external APIs. It has become so common, but we often fail to come up with a standardized solution.

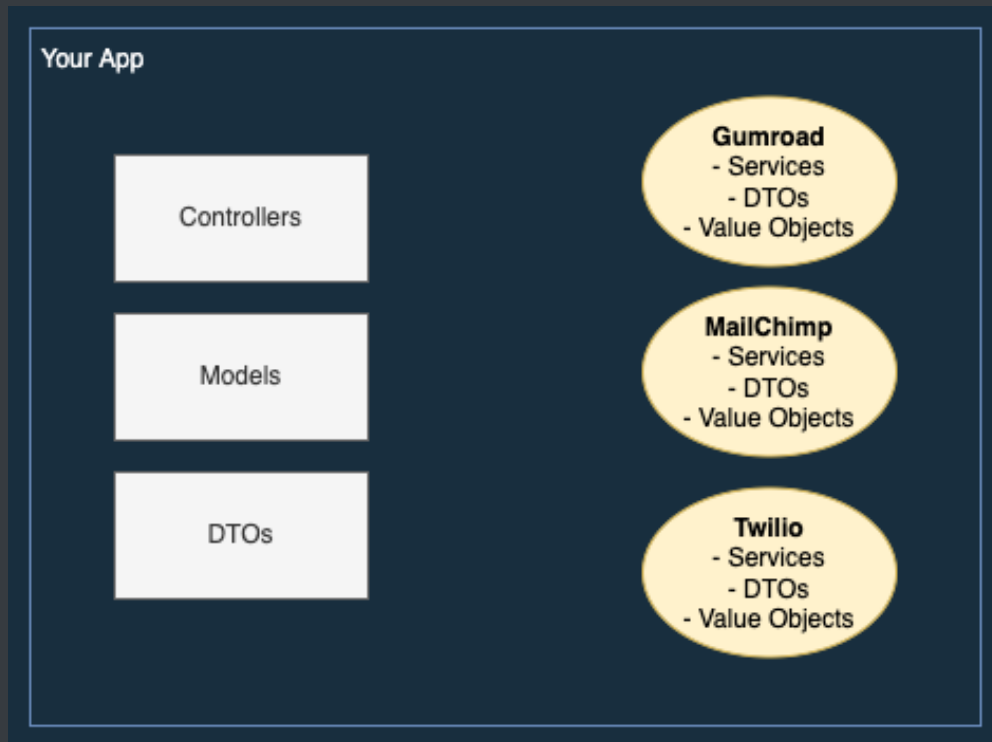
This is why I thought it'd be a fun project to show you. In this article, I'll write a Gumroad (e-commerce) SDK in three different ways:

- Using one service for the whole API. This is probably the common one.
- Having separate requests for each API endpoint.
- Using a package that solves some generic problems.

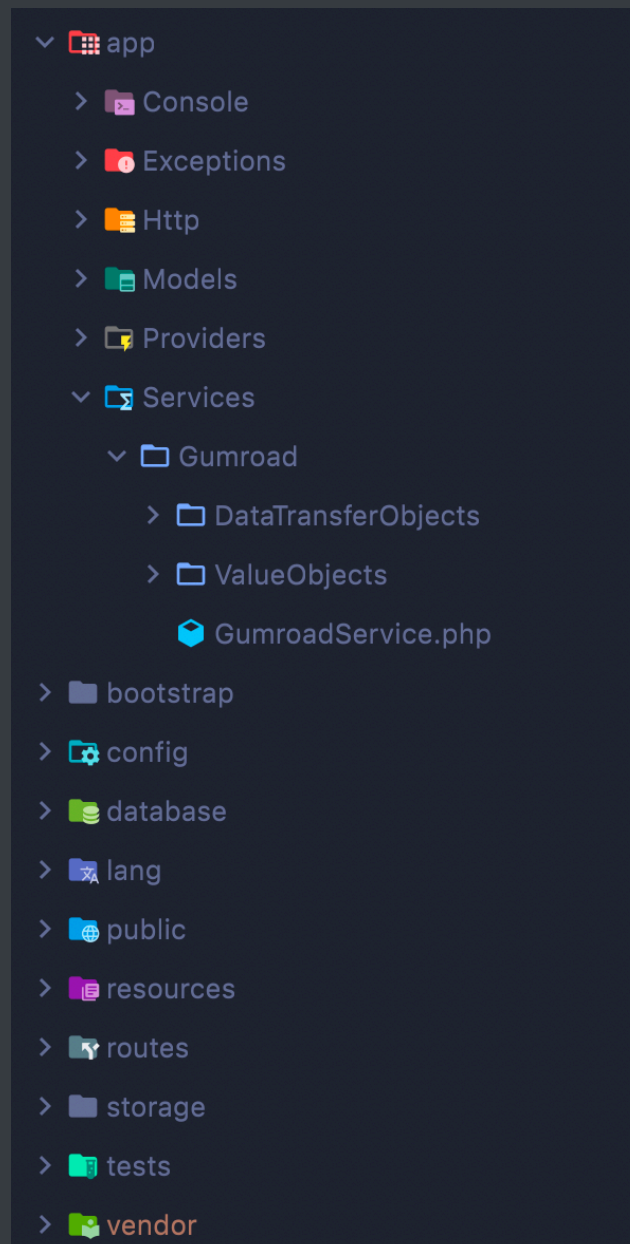
In a previous article, I already talked about APIs and used Gumroad API. However, this one will focus more on the structure of your project rather than the implementation of the SDK.

One Service

If there's only one takeaway from this book, it has to be this one: **treat your 3rd parties as if they were mini-applications inside your application.**



So each external API has its own namespace with DTOs, Value Objects, or Services if needed. This is how I implement this idea:



In the `Services` folder, I create a new folder for each 3rd party and treat it like it was a mini-application inside my app. In a minute, I'll show you what the `GumroadService` looks like, but first, let's take care of the configuration. There's a `config/services.php` where we can configure the 3rd parties:

```

return [
    'gumroad' => [
        'access_token' => env('GUMROAD_ACCESS_TOKEN'),
        'uri' => env('GUMROAD_URI'),
    ],
];

```

Usually, each service has its own service provider:

```

namespace App\Providers;

use App\Services\Gumroad\GumroadService;
use Illuminate\Support\ServiceProvider;

class GumroadServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->singleton(
            GumroadService::class,
            fn () => new GumroadService(
                config('services.gumroad.access_token'),
                config('services.gumroad.uri'),
            )
        );
    }
}

```

In this example, I use the singleton binding. This means Laravel will create one instance of the `GumroadService` class with the values in the config files and every time a developer type-hints `GumroadService` somewhere, the same singleton will be resolved from the container. It makes sense because there's no point in creating separate `GumroadService` instances.

And finally, `GumroadService` uses these config values:

```
namespace App\Services\Gumroad;

class GumroadService
{
    public function __construct(
        private readonly string $accessToken,
        private readonly string $uri,
    ) {}
}
```

Now, everything is ready to use this class:

```
class ProductController
{
    public function index(GumroadService $gumroad)
    {
        $gumroad->products();
    }
}
```

As I said earlier, the main focus will be the overall structure of our project, but let's quickly implement two endpoints.

Get all products:

```
namespace App\Services\Gumroad;

class GumroadService
{
    /**
     * @return Collection<ProductData>
     */
    public function products(): Collection
    {
        $products = Http::get(
            $this->url('products'),
            $this->query(),
        )->json('products');

        return collect($products)
            ->map(fn (array $data) => ProductData::fromArray($data));
    }
}
```

Get one product by ID:

```
namespace App\Services\Gumroad;

class GumroadService
{
    public function product(string $id): ProductData
    {
    }
```

```

        $product = Http::get(
            $this->url("products/$id"),
            $this->query(),
        )->json('product');

        return ProductData::fromArray($product);
    }
}

```

They are pretty simple. In almost all cases, I use a `query` and a `url` helper function to make things a bit cleaner:

```

private function query(array $extra = []): array
{
    return [
        'access_token' => $this->accessToken,
        ... $extra,
    ];
}

private function url(string $path): string
{
    return "{$this->uri}/$path";
}

```

They are quite straightforward. The last interesting thing about the `GumroadService` is that it uses a `ProductData` DTO:

```
namespace App\Services\Gumroad\DataTransferObjects;

use App\Services\Gumroad\ValueObjects\Price;

class ProductData
{
    public function __construct(
        public readonly string $name,
        public readonly Price $price,
        public readonly string $url,
    ) {}

    public static function fromArray(array $data): self
    {
        return new self(
            name: $data['name'],
            price: Price::fromCents($data['price']),
            url: $data['short_url'],
        );
    }
}
```

This is a plain PHP object, I don't use any packages. The interesting thing about Gumroad API (and a lot of other APIs) is that it returns the prices in cent value instead of the dollar. So if a product costs \$19 it will be 1900. For that reason I use a value object called `Price` :

```
namespace App\Services\Gumroad\ValueObjects;

class Price
{
    public function __construct(
        public readonly int $cents,
        public readonly float $dollars,
        public readonly string $formatted,
    ) {}

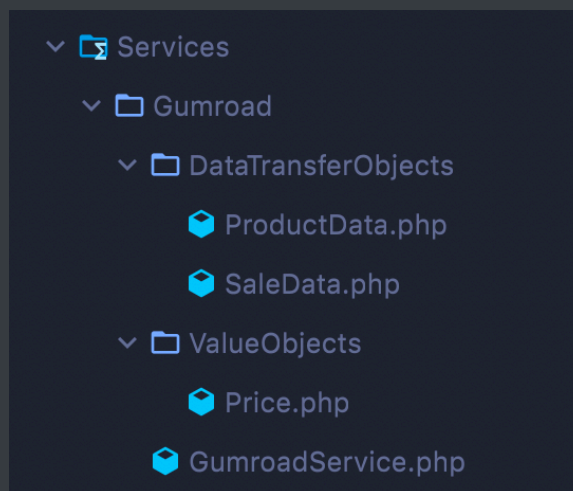
    public static function fromCents(int $cents): self
    {
        return new self(
            cents: $cents,
            dollars: $cents / 100,
            formatted: '$' . number_format($cents / 100, 2),
        );
    }
}
```

I can create a new `Price` object from the cent value and the result will be:

```
$price = Price::fromCents(1900);  
  
// Price object as an array  
[  
    'cents': 1900,  
    'dollars': 19.00,  
    'formatted': '$19.00',  
]
```

I find this approach very clean and high-level. And more importantly, it helps you avoid mistakes such as listing the cent value as it was in dollars, so a \$19 product becomes \$1900 on the frontend.

As you can see, all of these classes live inside the `Gumroad` namespace. This is what I meant by "mini-application":



Separate Requests

The first approach works very well in most situations. The only problem is when you need to implement 20 API endpoints. In this scenario you can easily end up with 500+ lines long class, overgeneralized functions, and nasty if-else statements to handle strange edge cases. So it's easy to end up with a big, hard-to-maintain SDK class.

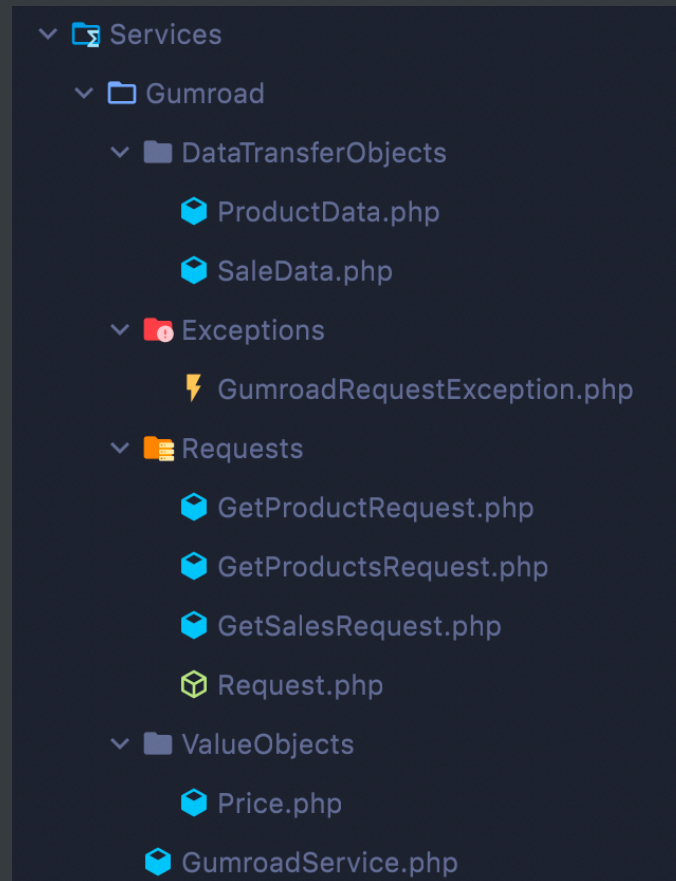
One approach to solve this problem is to treat 3rd party requests as if they were your own `FormRequests`. When we are dealing with our own requests it's a standard to create a separate class for each one:

```
class ProductController
{
    public function index(GetProductsRequest $request)
    {
        // ...
    }
}

class GetProductsRequest extends FormRequest
{
    // ...
}
```

Here's my question: why not do the same thing with 3rd party requests?

This is what the structure would look like:



In the Requests folder we have classes like:

- GetProductRequest
- GetProductsRequest
- GetSalesRequest

Just as they were standard form requests. Let's take a look at one of these requests:

```

namespace App\Services\Gumroad\Requests;

class GetProductsRequest extends Request
{
    public function send(): Collection
    {
        $data = Http::get(
            $this->url('products'),
            $this->query(),
        )->throw();

        if (!$data['success']) {
            throw GumroadRequestException::unknownError(
                Arr::get($data, 'message', '')
            );
        }

        return collect($data->json('products'))
            ->map(fn (array $data) => ProductData::fromArray($data));
    }
}

```

This looks almost the same as the `products` method earlier. The same is true for the `GetProductRequest` :

```
namespace App\Services\Gumroad\Requests;

class GetProductRequest extends Request
{
    public function send(string $id): ProductData
    {
        $data = Http::get(
            $this->url("products/$id"),
            $this->query(),
        )->throw();

        if (!$data['success']) {
            throw GumroadRequestException::productNotFound($id);
        }

        return ProductData::fromArray($data->json('product'));
    }
}
```

As you can see, I still use the `url` and `query` helpers, but now they are in the base `Request` class:

```

namespace App\Services\Gumroad\Requests;

abstract class Request
{
    public function __construct(
        protected readonly string $accessToken,
        protected readonly string $uri,
    ) {}

    protected function query(array $extra = []): array
    {
        return [
            'access_token' => $this->accessToken,
            ... $extra,
        ];
    }

    protected function url(string $path): string
    {
        return "{$this->uri}/$path";
    }
}

```

Since Laravel already has more than one class called `Request` you can easily call this one `GumroadRequest` so it's not confusing. These requests can be used in two ways.

The "stand-alone" version:

```
$products = app(GetProductsRequest::class)->send();
```

The injected version:

```
class ProductController
{
    public function index(GetProductsRequest $request)
    {
        $products = $request->send();
    }
}
```

However, the first one looks a bit weird to me, while the second one is just confusing. We usually inject `FormRequests` in methods. Now we have two kinds of requests. It's not optimal, in my opinion.

To solve these problems we can keep the `GumroadService` class as an entry point to access these requests:

```
namespace App\Services\Gumroad;

class GumroadService
{
    /**
     * @return Collection<SaleData>
     */
    public function sales(?Carbon $after = null): Collection
    {
        return app(GetSalesRequest::class)→send($after);
    }

    /**
     * @return Collection<ProductData>
     */
    public function products(): Collection
    {
        return app(GetProductsRequest::class)→send();
    }

    public function product(string $id): ProductData
    {
        return app(GetProductRequest::class)→send($id);
    }
}
```

In this class, you can use either the `app` function to resolve the requests, or you can inject them into the controller. I choose the first one because injecting 15 classes into the constructor looks weird.

This class can be used as usual:

```
class ProductController
{
    public function index(GumroadService $gumroad)
    {
        $products = $gumroad->products();
    }
}
```

There's one drawback of this solution. We don't need to bind a `GumroadService` instance anymore in the service provider. Now the config values are consumed by request classes. I have a base `Request` class that accepts the access token and base URL in the constructor. However, in `GumroadService` I type-hint and inject the subclasses. For that reason, I need to bind these values for every subclass in the `GumroadServiceProvider` :


```

namespace App\Providers;

class GumroadServiceProvider extends ServiceProvider
{
    public function register()
    {
        $requests = [
            GetProductsRequest::class,
            GetProductRequest::class,
            GetSalesRequest::class,
        ];

        foreach ($requests as $requestClass) {
            $this->app->singleton(
                $requestClass,
                fn () => new $requestClass(
                    config('services.gumroad.access_token'),
                    config('services.gumroad.uri')
                )
            );
        }
    }
}

```

So you need to add every request to the `$requests` array. You can make it dynamic with some `Reflection` magic, but it's still extra work.

Transporter by JustSteveKing

Using separate requests is a nice approach, in my opinion. However, there are some problems we ran into:

- Binding every request individually
- Using the `app` method to resolve the requests
- Having a base `Request` class with generic helpers
- Repeating the same `Http` calls inside requests

Fortunately, [@JustSteveKing](#) also ran into these problems and solved them with a package called [laravel-transporter](#).

This is what a simple request looks like with transporter:

```
namespace App\Services\Gumroad\Requests;

class GetProductsRequest extends GumroadRequest
{
    protected string $method = 'GET';
    protected string $path = 'products';
}
```

Much less noise, right? And this is how it can be used from `GumroadService` :

```
class GumroadService
{
    /**
     * @return Collection<ProductData>
     */
    public function products(): Collection
    {
```

```

return GetProductsRequest::build()
    →send()
    →throw()
    →collect('products')
    →map(fn (array $data) => ProductData::fromArray($data));
}
}

```

The `send` method returns an `Illuminate\Http\Client\Response` so we can use every Laravel helper, such as `collect`.

As you can see the `GetProductsRequest` extends the `GumroadRequest` class. This is where I add the access token to every request:

```

namespace App\Services\Gumroad\Requests;

class GumroadRequest extends Request
{
    public function __construct(HttpFactory $http)
    {
        parent::__construct($http);

        parent::withQuery([
            'access_token' =>
config('services.gumroad.access_token'),
        ]);
    }
}

```

Earlier I used the `query` method in every request to do the same. Now, I don't need to worry about it.

The only challenge with `laravel-transporter` is appending variables (such as an ID) to the URL. As far as I know, this is how you can do it:

```
namespace App\Services\Gumroad\Requests;

class GetProductRequest extends GumroadRequest
{
    protected string $method = 'GET';
    protected string $path = 'products/%s';

    public function withProductId(string $productId): self
    {
        return $this->setPath(sprintf($this->path(), $productId));
    }
}
```

And this is how you can use the `withProductId` method when sending the request:

```

class GumroadService
{
    public function product(string $id): ProductData
    {
        $data = GetProductRequest::build()
            →withProductId($id)
            →send()
            →throw()
            →json('product');

        return ProductData::fromArray($data);
    }
}

```

It could be a bit better, but this works just fine.

And the last piece of the puzzle is the configuration. Transporter only needs a `base_uri` in order to know what URL to use when sending a request. We have a `config/transporter.php` file:

```

return [
    'base_uri' => env('GUMROAD_URI'),
];

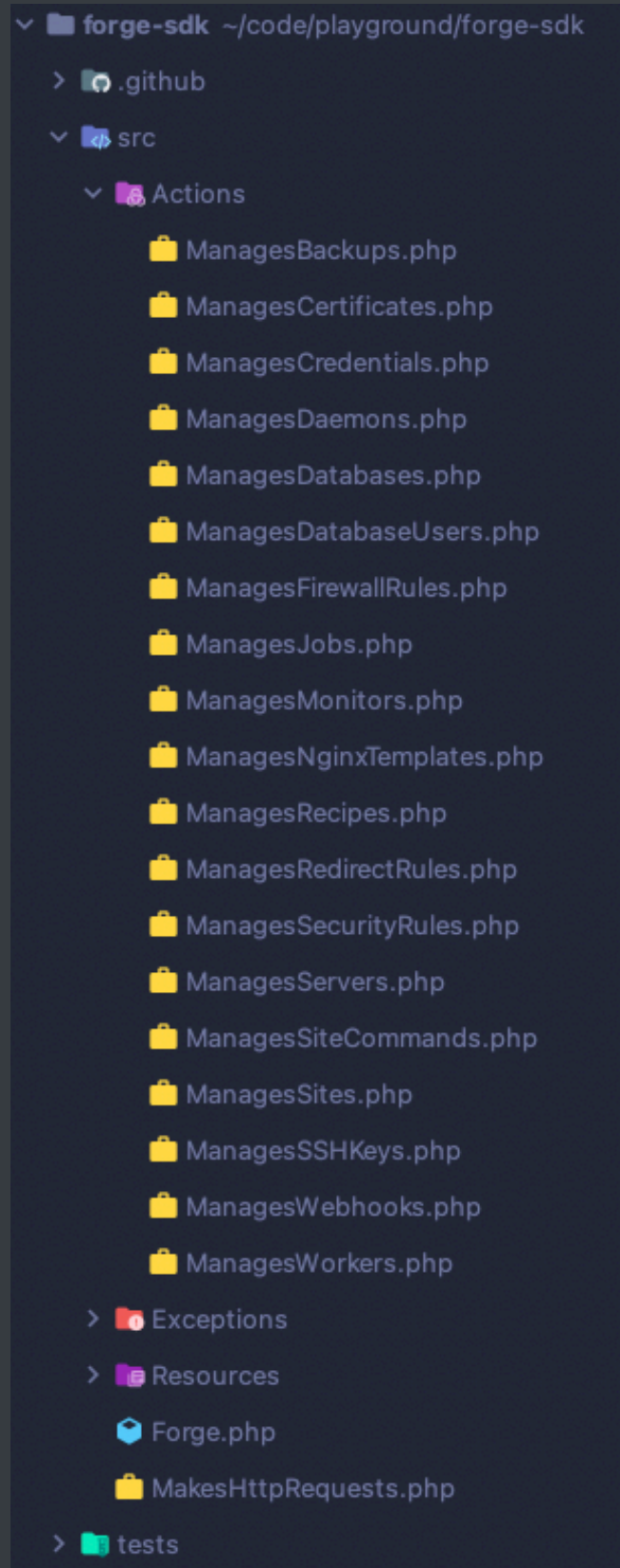
```

By default, you can only have one `base_uri` so it assumes that you have only one 3rd party in your application. However, it can be overwritten via the `$baseUrl` variable in the `base Request` class. For example, if you have to write an SDK for MailChimp you can do this:

Laravel Forge SDK

Another great example of separating your requests is Laravel Forge SDK. You can find the repo here: <https://github.com/laravel/forge-sdk>

If you don't know, Forge is a web-based service to manage servers and deployments. Forge SDK is a wrapper for the Forge API. If you take a look at the source code you'll find something like this:



It has 19 classes inside the `Actions` namespace. Each of them takes care of one type of request, for example, `ManagesServers` has methods like:

- `servers` returns all servers
- `server($serverId)` returns one server

- `createServer` creates a new server

So it's not really one class for every request, but rather one class for one resource. It's also an excellent approach and you can handle big APIs with fewer classes, but still, your classes remain small.

However, these "actions" are in fact, traits not classes. There's a simple class called `Forge` that uses all of these traits:

```
class Forge
{
    use MakesHttpRequests,
        Actions\ManagesJobs,
        Actions\ManagesSites,
        Actions\ManagesServers,
        Actions\ManagesDaemons,
        Actions\ManagesWorkers,
        Actions\ManagesSSHKeys,
        ... ;
}
```

Using this approach you still have a single entry-point class to access every request:

```
$forge = new Forge();

$servers = $forge->servers();
$server = $forge->server('abc-123');
$forge->createServer([ ... ]);
```

This is very similar to what we did earlier. In fact, it's almost an identical approach:

- Each action is a request from the previous examples
- The `Forge` class is the equivalent of the `GumroadService`

By the way, if you like the idea of traits, you can do the same in your own SDKs.

I think writing separate requests for each API endpoint is a great approach, especially if you're writing an SDK that interacts with 10+ endpoints (or in general, it's big). By using a package like `laravel-transporter`, you can eliminate a lot of boilerplate and generic code.

Custom Fields

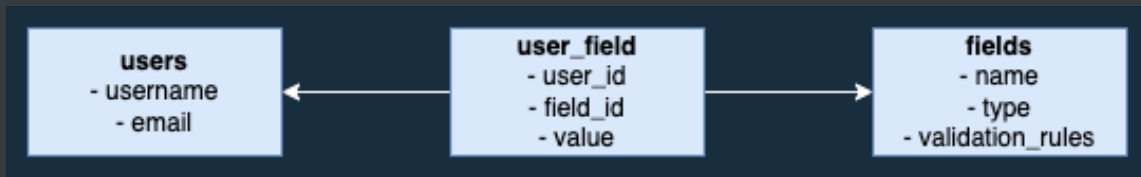
I've been working on an application where we have custom fields for users. Here's the idea:

- The `users` table has only a few columns. Things like: `username`, and `password`.
- There's a `fields` table where there are additional custom fields, such as `full_name`, `company_email`, `personal_email`, and so on.
- Every client decides what fields they want to use and sets up the system accordingly.

This part of the project is legacy, and it has some flaws, so I thought it'd be a good idea to write about this topic.

Data Modeling

The database is quite simple:



This is what the `fields` table looks like:

name	type	validation_rules
full_name	string	["string"]
phone	number	["numeric", "digits:11"]
join_date	date	["date_format:Y-m-d"]

The `type` column describes the basic type of the field. It can be used by the frontend to render the appropriate HTML element:

- `full_name` will be a simple `input` element
- `phone` will be an `input` with `type="number"`
- `join_date` will use your favorite date picker plugin (it's just a date without time)

The `validation_rules` will be used by the backend. When creating a new user we're going to validate the values against these rules. To have a better idea, this is what the `POST /users` request looks like:

```
{
  "username": "martin.joo",
  "email": "martin@martinjoo.dev",
  "fields": [
    {
      "id": 1,
      "value": "Martin Joo"
    },
    {
      "id": 2,
      "value": "3630111111"
    }
  ]
}
```

The first field is the `full_name` and the second one is the `phone`. The backend will validate these values against the rules defined in the `validation_rules` column.

That being said, let's create the models and set up the relationships.

```

namespace App\Models;

class Field extends Model
{
    use HasFactory;

    protected $casts = [
        'validation_rules' => 'array',
    ];
}

```

The `Field` model doesn't have any method only a cast. Validation rules are stored as JSON strings, so we can cast them to PHP arrays.

The `User` model:

```

namespace App\Models;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    public function fields(): BelongsToMany
    {
        return $this->belongsToMany(Field::class, 'user_field')
            ->withPivot('value')
            ->withTimestamps();
    }
}

```

It's also pretty simple. It defines a `fields` method which is a many-to-many relationship. A user can have many fields, but a field belongs to many users. Some notes:

- The second parameter of the `belongsToMany` method is the table name. By default, Laravel will look for a `field_user` table (alphabetical order), but I prefer the name `user_field`.
- `withPivot` means we want to see the `value` column from `user_field` whenever we query the fields of a user.
- `withTimestamps` means that Laravel will update the `created_at` and `updated_at` columns' values in the `user_field` table.

This will result in the following:

```
Field.php x 2022_06_05_080343_create_fields_table.php x User.php x
>>> $user->fields
=> Illuminate\Database\Eloquent\Collection {#4467
    all: [
        App\Models\Field {#4447
            id: 1, 'email_verified_at' => 'datetime',
            name: "full_name",
            type: "string",
            validation_rules: "["string"]",
            created_at: "2022-06-05 09:01:50",
            updated_at: "2022-06-05 09:01:50",
            pivot: Illuminate\Database\Eloquent\Relations\Pivot {#4463
                user_id: 4,
                field_id: 1,
                value: "Martin Joo",
                created_at: "2022-06-05 09:17:19",
                updated_at: "2022-06-05 09:17:19",
            },
        ],
    ],
}
```

Please note the `value` attribute inside the `pivot` . It's available because of the `withPivot` .

User API

Let's start with creating and updating users. This is the request class:

```
namespace App\Http\Requests;

class UpsertUserRequest extends FormRequest
{
    public function rules()
    {
        return [
            'username' => ['required'],
            'email' => [
                'required',
                'email',
                Rule::unique('users', 'email')->ignore($this->user),
            ],
            'password' => ['required'],
            'fields' => ['nullable', 'sometimes', 'array'],
            'fields.*.id' => ['required', 'exists:fields,id'],
            'fields.*.value' => ['required'],
        ];
    }
}
```

There are three interesting things:

- I always use the name `upsert` for requests, actions, and controller methods that take care of both inserting and updating a model. You'll see more examples of this later.
- The e-mail address must be unique but we have to ignore the current user's e-mail. Otherwise, the update would fail, because the e-mail already exists in the database.


```

namespace App\DataTransferObjects;

class UserData
{
    public function __construct(
        public readonly ?int $id,
        public readonly string $username,
        public readonly string $email,
        public readonly string $password,
    ) {}

    public static function fromRequest(
        UpsertUserRequest $request
    ): self {
        return new self(
            id: $request->user?->id,
            username: $request->username,
            email: $request->email,
            password: $request->password,
        );
    }
}

```

This is just a plain PHP object that contains the attributes of a user. In this tiny example, there's not much benefit to using DTOs. However, in larger applications, it's a very good idea, in my opinion. If you're not familiar with them, [here's an article](#) where you can read more about them.

We can also create a DTO for the field values:


```
"fields": [
  {
    "id": 1,
    "value": "Martin Joo"
  },
  {
    "id": 2,
    "value": "3630111111"
  }
]
```

Each object will be a `FieldValueData` instance. So this class simply helps us avoid working with random associative arrays later. Instead, we'll have some pretty simple objects with type-hinted properties.

Now we can move on to the controller:

```
namespace App\Http\Controllers;

class UserController extends Controller
{
    public function store(UpserUserRequest $request)
    {
        return $this->upsert($request);
    }

    public function update(
        UpserUserRequest $request,
        User $user
    ) {
```

```

        return $this->upsert($request);
    }
}

```

Both `store` and `update` use a method called `upsert` :

```

private function upsert(
    UpsertUserRequest $request
): UserResource {
    $user = UpsertUserAction::execute(
        UserData::fromRequest($request),
        FieldValueData::collect($request->fields),
    );

    return UserResource::make($user);
}

```

This method creates DTOs from the request and calls the action (we'll see it in a minute). Some important notes:

- `upsert` is the only method I write inside controllers other than the basic CRUD functions! And if you think about it, it's also a resource method.
- The `id` property the `UserData` is nullable. This allows the `UpsertUserAction` to easily handle both creating and updating users with the same code.

Here's the action:

```

class UpsertUserAction
{
    /**
     * @param Collection<FieldValueData> $fieldValues
     */
    public static function execute(
        UserData $userData,
        Collection $fieldValues,
    ): User {
        // This is where we're going to validate the fields later

        $user = User::updateOrCreate(
            [
                // It's either NULL or a valid ID
                'id' => $userData->id,
            ],
            [
                'username' => $userData->username,
                'email' => $userData->email,
                'password' => bcrypt($userData->password),
            ],
        );

        $user->fields()->detach();

        foreach ($fieldValues as $fieldValue) {
            $user->fields()->attach($fieldValue->id, [
                'value' => $fieldValue->value,
            ]);
        }
    }
}

```

```
        return $user;
    }
}
```

As you can see, there are two parts to this function:

- It saves the user using the `updateOrCreate`. This method accepts two arrays. The first one will be used in a `select` query. If Eloquent finds a record based on this array it will execute an `update` query. Otherwise, it'll run an `insert`. When creating a new user, the ID is going to be NULL in the `UserData` class, so it'll run an insert query.
- After the user is created it detaches every field and attaches the ones from the request.

There's another important part I left out from the action. We need to validate the fields using the validation rules from the database. Since it's not really strongly related to creating users and we will need it in another API later, we can create a separate action.

The `ValidateFieldAction` will validate one field:


```

class ValidateFieldAction
{
    /**
     * @throws ValidationException
     */
    public static function execute(
        Field $field,
        string $value,
    ): void {
        $validator = Validator::make(
            [
                $field->name => $value,
            ],
            [
                $field->name => $field->validation_rules,
            ],
        );

        if ($validator->fails()) {
            throw ValidationException::withMessages(
                $validator->errors()->toArray()
            );
        }
    }
}

```

It accepts a `string $value` that comes from the request, and a `Field` instance that we can query later in the `UpsertUserAction`. We need to create a validator object manually. It accepts the following arrays:

```
Validator::make(
    [
        'phone' => '36301111111',
    ],
    [
        'phone' => ['numeric', 'digits:11'],
    ],
);
```

The first array contains the data, meanwhile, the second one contains the validation rules:

```
Validator::make(
    [
        $field->name => $value,
    ],
    [
        $field->name => $field->validation_rules,
    ],
);
```

Since there's an `array` cast in the `Field` model, the `validation_rules` will be a valid array instead of a string. If the validator fails, we simply throw a `ValidationException`.

The last piece of the puzzle is to use this action from the `UpsertUserAction`:

```

namespace App\Action;

class UpsertUserAction
{
    /**
     * @param Collection<FieldValueData> $fieldValues
     */
    public static function execute(
        UserData $userData,
        Collection $fieldValues,
    ): User {
        $fields = Field::whereIn(
            'id',
            collect($fieldValues)→pluck('id')
        )→get();

        foreach ($fieldValues as $fieldValue) {
            $field = $fields→where('id', $fieldValue→id)→first();

            ValidateFieldAction::execute($field, $fieldValue→value);
        }
    }
}

```

As the first step, I query every field. I do this because it prevents us from running into N+1 query problems. This way the `ValidateFieldAction` doesn't need to query fields at all. Other than that, it just iterates through the field values and calls the validate action for each field.

Since a lot is happening inside the `UpsertUserAction` it's a good idea to use DB transactions:

```
namespace App\Action;

class UpsertUserAction
{
    /**
     * @param Collection<FieldValueData> $fieldValues
     */
    public static function execute(
        UserData $userData,
        Collection $fieldValues,
    ): User {
        return DB::transaction(function () use (
            $userData,
            $fieldValues
        ) {
            $fields = Field::whereIn(
                'id',
                collect($fieldValues)→pluck('id')
            )→get();

            foreach ($fieldValues as $fieldValue) {
                $field = $fields
                    →where('id', $fieldValue→id)
                    →first();

                ValidateFieldAction::execute($field, $fieldValue→value);
            }
        });
    }
}
```


User Field API

We can also create a separate API for creating and updating individual fields for a user. The endpoint for adding a field to a user is this:

```
POST /users/1/fields/1
```

The request looks like this:

```
{  
  "value": "Martin Joo"  
}
```

This means we want to add a `full_name` field to this particular user with this value. Since there's already a `ValidateFieldAction` we can simply reuse it:

```
namespace App\Http\Controllers;  
  
class UserFieldController extends Controller  
{  
    public function store(  
        Request $request,  
        User $user,  
        Field $field  
    ) {  
        ValidateFieldAction::execute(  
            $field,  
            $request->value,  
        );  
    }  
}
```

```

        $user->fields()->attach($request->id, [
            'value' => $request->value,
        ]);

        return UserResource::make($user->load('fields'));
    }
}

```

First, it validates the field's value then it attaches it to the user. The `update` looks pretty similar to this one:

```

public function update(
    Request $request,
    User $user,
    Field $field
) {
    ValidateFieldAction::execute(
        $field,
        $request->value,
    );

    $user->fields()->updateExistingPivot($field->id, [
        'value' => $request->value,
    ]);

    return UserResource::make($user->load('fields'));
}

```

The validation is exactly the same, but now, instead of `attach` I use the `updateExistingPivot`. It will look for a record in the `user_field` table that matches the ID given in the first parameter and updates it with the array in the second one.

Partly, this is why I extracted the `ValidateFieldAction` into a separate class. It's easy to reuse now. However, if we don't need the user field API, it's still a good idea to have a separate class, in my opinion.

As you can see, it's a bit weird that both the `store` and `update` methods have the same arguments. It's because this controller is not a "classic" resource controller. It does not use the IDs from the `user_field` table but the `user.id` and `field.id`. Another approach would be to introduce a `UserField` model and then have two methods such as:

```
public function store(Request $request, User $user);

public function update(Request $request, User $user, UserField $userField);
```

And then the API endpoints should look like this:

```
POST /users/1/fields

PATCH /users/1/fields/113
```

Where `113` refers to a record in the `user_field` pivot table. But in this particular case, I went with this API:

```
POST /users/1/fields/2

PATCH /users/1/fields/2
```


The biggest downside of this is that I'm deviating from REST API standards. The advantages:

- It doesn't require a `UserField` model
- The controller is pretty simple, we can easily reuse every class
- And maybe in this case, it's a bit more understandable because, on the API level, we're only talking about `Field` IDs but not `UserField` IDs. If we have both of them, it can cause some confusion.

At the end of the day, it's a personal (or team) preference, so go with the one that works best for you.

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](#)