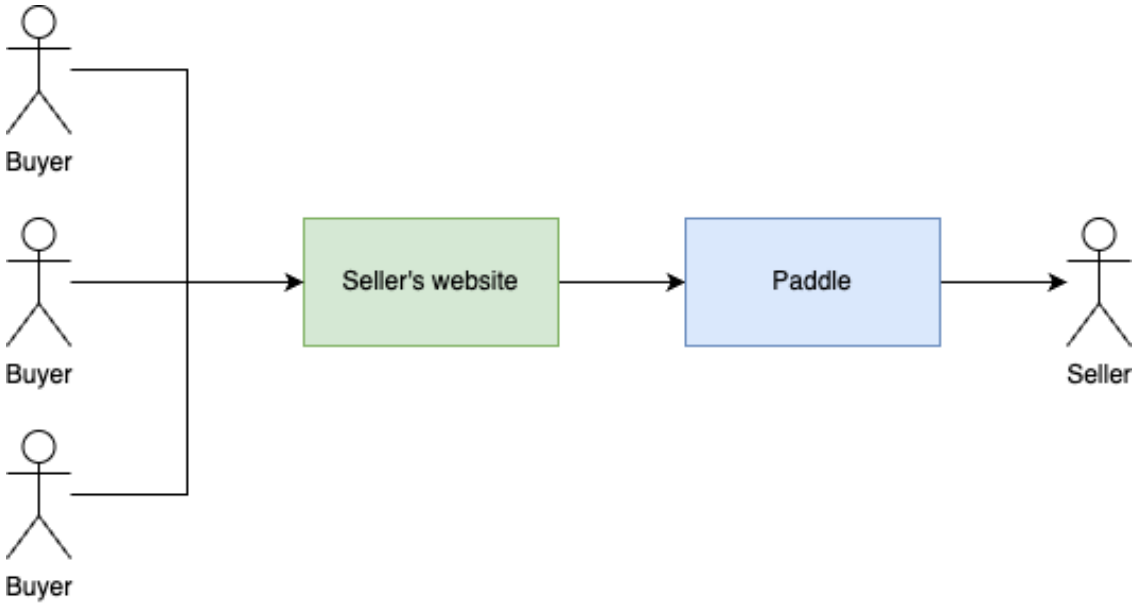
BUILDING FINANCE APPS WITH LARAVEL

MARTIN JOO

Introduction

In this book, I'd like to show you some ideas and tips that you can use in any financial-related applications of features. We're going to rebuild some features of Paddle.

Paddle is a finance application. More specifically it's a merchant of record that looks like this:



This is what's happening:

- A seller (or content creator) uploads a product to Paddle
- They integrate Paddle into their landing page
- Buyers buy the product from the landing page using a Paddle checkout form
- Paddle pays the seller every month

I personally use Paddle to sell my books and SaaS and it's a great service. The main benefit is that you don't have to deal with hundreds or thousands of invoices and VAT ramifications. Paddle handles it for you. They send an invoice to the buyer and apply the right amount of VAT based on the buyer's location. They also handle VAT ramifications. You, as the seller, don't have to deal with any of that stuff. They just send you the money once every month and you have only one invoice. It also provides nice dashboards and reports.

In this short book, I'd like to rebuild some of its core features. My experience is that there are lots of (mostly) legacy projects where dealing with money, numbers, and payments is implemented in a horrible way. Float numbers, strings, integers, calculation errors because of the decimals, inconsistency all over the place, `$29.99998` and `30.000001%` on the frontend. You know what I mean.

These are the main features we're replicating:

- Buying something from the seller. This is called a transaction.
- Triggering a webhook. Sellers can set up webhooks such as "call this URL when someone buys a product."
- Exporting transaction reports.
- Paying out sellers every month in a scheduled way based on their past transactions.

And of course, some other, smaller features as well, but this is the core functionality of our little Paddle clone.

There are two other non-functional requirements for this project:

- Handling money and numbers in a reliable way.
- Trying to think about and handle the possible errors. So the whole application should be reliable in general because we're dealing with money.

Let's think about what reliability means.

Dealing with money

In order to handle money values in a reliable way we need to store them as cent values. Instead of storing $19.99 as `19.99` in the database, we store it as `1999`. This way we'll have no problem with decimal points, rounding, multiplying, or dividing numbers.

The other thing we can do is to use value objects instead of treating them as float numbers in PHP. So instead of this:

```
$earnings = $transaction->amount - $fee - $tax;
```

We'll write code such as this:

```
$earnings = $transaction->amount->subtract($tax)->subtract($fee);

return $earnings->getAmount();
```

sWe'll use Eloquent attribute casts and value objects to make everything as smooth as possible.

If you're not sure what a value object is, check out this article. Essentially, it's just an object that represents a value such as a money amount, a date, or a percentage. It doesn't have behavior (methods) only some factory functions.

Dealing with dates

Another thing I find quite confusing in features such as dashboards, and reports is dealing with dates. Have you ever seen code like this?

```
public function getRevenue($params)
{
  // ...
  $params['startDate']
  $params['endDate']
}
```

`startDate` and `endDate` are PHP Date objects.

The other class in the same project does this:

```
public function getTransactionsByUser($user, $dates)
{
  // ...
  $dates['from']
  $dates['to']
}
```

`from` and `to` are strings.

The third method uses dedicated `$fromDate` and `$endDate` variables and they are UNIX timestamps. In the fourth class... You know what I'm talking about. It's a complete mess.

We're going to solve that problem by introducing domain-specific objects such as `StartDate` end `EndDate` and we'll have functions such as:

```
public function revenue(DateInterval $interval, User $user): Money
{
  $sum = $this
    ->whereUser($user)
    ->whereBetween('created_at', [
      $interval->startDate->date,
      $interval->endDate->date,
    ])
    ->sum('revenue');

  return Money::USD($sum);
}
```

`$interval->startDate` is an instance of `StartDate` which is **always** (no matter what) `Y-m-d 00:00:00` and `$internal->endDate` is an instance of `EndDate` which is **always** (no matter what) `Y-m-d 23:59:59`. In Paddle, the smallest unit you can use when using their reports is days. So a report always shows data from `00:00:00` to `23:59:59`. We'll have a central object that handles this logic.

Queue jobs

We're going to use queue jobs for a number of things:

- Sending notifications
- Paying out sellers
- Cleaning up old exports to save disk space
- Triggering webhooks

Lots of Laravel projects have only one queue (the `default`). It can be perfectly fine. However, just think about it. What happens if we need to do the following things at the same time:

- Sending out 5000 notifications. Let's say each of them takes 0.5s
- Triggering 2000 webhooks. Each of them takes 1s

And then, suddenly, you need to pay out 20,000 sellers using the same queue. Now you have to wait: `5000*0.5 + 2000*1` so `4500s` which is 75 minutes. Of course, it's not the end of the world, but what happens if you need to trigger 20,000 webhooks instead of 2k? The number goes up to 6 hours. You have to wait 6 hours before paying out your customers. The payout job should take priority no matter what. And don't forget, if you're sending 5000 notifications the first webhook trigger is going to happen after 2500s. We'll handle these problems by using multiple queues.

Models

Let's think about the database structure first. We need products to sell and transactions to keep a record of the sales:

id	uuid	title	price	url	user_id
1	9b058043-763c-46a3-ad28-33a013f03a1f	Domain-Driven Design with Laravel	3900	https://domain-driven-design-laravel.com/	1
2	9b058046-145d-4a83-aa1e-02f6baad708c	DevOps with Laravel	3900	https://devops-with-laravel.io/	1

`3900` means `3900` cents which is $39. As I said earlier, I'm going to store prices as integers in cent values.

Every table has an `id` and a `uuid` column as well. The reason is this:

- An auto-increment integer value is the best option for MySQL to use in foreign key constraints, `join` statements, or in BTREEs. This is the fastest ID column type by design.
- But I don't want to expose integer IDs in API URLs because it's a security risk. It exposes (at least) two problems
  - The German tank problem. So in WW2 Germans built lots of tanks and the Allies wanted to know how many. Germans used sequential serial numbers such as an auto-increment ID. When the Allies captured a German tank they were able to guess how many did the Germans build in a month based on the serial number. When you expose integer IDs you reveal information about your data. It may or may not be a problem in your specific case. If we expose product IDs we publicly reveal the number of products in the application. Probably not a big problem. But what if we also expose the ID of transactions by a user? For example: `/api/users/1/transactions/9` We just "revealed" some business or revenue-related information about user `1`.
  - Enumeration attack problems. If you make an authorization mistake and you use integer IDs in your API, you're in trouble. Imagine that you forgot to authorize the `DELETE /api/products/{id}` endpoint in your `Request` class. Now anyone can wipe your products table by just running this:

```
#!/bin/bash
for ((i=1; i<=100; i++)); do
  curl -X DELETE "https://myawesomeapp.com/api/products/$i"
done
```

This is why I'm using both integer IDs and UUIDs. Auto-increment IDs are being used internally in database queries, but only UUIDs are exposed via APIs.

Now let's take a look at the `transactions` table:

id	uuid	product_id	quantity	revenue	balance_earnings	product_data	payout_id	stripe_id
1	9b058043-55fe-4bc7-be5c-50baaf583a98	1	1	3900	3120	{"id": 1, "url": "...", "uuid": "...", "title": "..."}	NULL	acac83e2-5a44-3e27-a588-06f86ce202e6
2	9b058046-37e6-48e1-997c-aeaae908e96d	2	1	3900	3120	...	NULL	...

A few things to consider:

- There's a between `revenue` and `balance_earnings`. Paddle takes 5% from every transaction and your beloved country takes another 5%-27% in the name of VAT. By the way, Hungary (my country) has the highest VAT in the world with 27%. So there are 4 additional columns in the table:
  - `fee_rate` and `fee_amount`. The first one is a percentage and the second one is a dollar amount.
  - `tax_rate` and `tax_amount`.
  - `balance_earnings = revenue - tax_amount - fee_amount`
- `product_data` is a JSON column that stores the whole `Product` object at the moment of the sale. Of course, we store the `product_id` as a foreign key, but we need to store the original product because products can be changed. If product 1 had a price of $39 at the time of the transaction but now it costs $59 it can cause some trouble later. By storing everything in the `product_data` column we always have the source of truth. However, we still need the `product_id` to write optimal queries, such as exporting all transactions by a given product. When you have millions of rows, you probably don't want to use an unindexed JSON column in a `join` statement.
- When a transaction happens we're going to invoke Stripe API and we'll get an ID in response. This is the `stripe_id` column.
- We're going to talk more about `payout_id` later.
- The table also has the following columns: `customer_email`, `user_id`, `created_at`

Money value object

Before writing any business logic we need to take care of money values. There's an excellent package called money that gives us some value objects to deal with money and currencies.

It works like this:

```
$fiveUsd = Money::USD(500);

$fourUsd = Money::USD(400);

$nineUsd = $fiveUsd->add($fourUsd);

// 900
echo $nineUsd->getAmount();
```

It treats values as integers in cent values, and everything is an object so you add to numbers by calling the `add` method. We need to cast the `Product` model's `price` attribute to this `Money` object. In order to do that we need a custom cast:

```
<?php

namespace App\Casts;

use Money\Money;

class MoneyCast implements CastsAttributes
{
  public function get(Model $model, string $key, mixed $value, array
$attributes): mixed
  {
    return Money::USD($value);
  }

  public function set(Model $model, string $key, mixed $value, array
$attributes): mixed
  {
    return $value->getAmount();
```

```
    }
}
```

There are only two methods:

- `get` handles when you have a model and you want to access the property. In this case, the `$value` attribute is the value of the `price` column from the database. So this method runs when you do this: `echo $product->price`
- `set` handles when you want to set a property on the model. But instead of `$product->price = 500` we want to do this: `$product->price = Money::USD(500)`. This is why the `set` method returns with `$value->getAmount()`. `$value` is a `Money` object and the `set` method gets the amount so `500` will be inserted into the database.

To sum it up, we want something like this:

```php
$product->price = Money::USD(500);

// 500 as an integer is stored in the DB
$product->save();

// It's a Money\Money object
echo $product->price;

// 500
echo $product->price->getAmount();
```

I'm not proud of this sentence, but here's the truth: we made the whole thing a little bit more like Java. Which is a good thing if we're dealing with money and transactions.

The last thing is to actually use the `MoneyCast`:

```php
namespace App\Models;

use App\Casts\MoneyCast;

class Product extends Model
{
  use HasFactory;
  use SoftDeletes;

  protected $fillable = [
    'title',
    'price',
    'url',
  ];

  protected $casts = [
    'price' => MoneyCast::class,
  ];

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

We need to specify that the `price` column uses our shiny `MoneyCast`.

Buying a product (transaction)

When someone buys something we need to create a new transaction so the API endpoint is `POST /transactions`. A transaction needs the following three parameters:

- product_id
- quantity
- customer_email

The request looks like this:

```
class StoreTransactionRequest extends FormRequest
{
  public function rules(): array
  {
    return [
      'product_id' => ['required', 'exists:products,id'],
      'quantity' => ['required', 'numeric', 'min:1'],
      'customer_email' => ['required', 'email'],
    ];
  }

  public function product(): Product
  {
    return Product::findOrFail($this->product_id);
  }

  public function quantity(): int
  {
    return (int) $this->quantity;
  }

  public function customerEmail(): string
  {
    return $this->customer_email;
  }
}
```

As you can see, I like these pretty straightforward getter methods in request classes. I don't write business logic here, only one-liner `findOrFail` type queries to get something from the database. It's highly subjective, of course. In theory, you shouldn't do something like that, but in practice, I never experienced any problem with this approach.

Now let's think about what we need to do when someone buys a product:

- Calculate the total price based on the product and the quantity
- Calculate the VAT amount based on the customer's IP address
- Charge the customer using Stripe
- Store the actual transaction
- Trigger webhooks

In this chapter, we're not going to deal with Stripe. We'll handle that later.

```
class TransactionController
{
  public function store(StoreTransactionRequest $request, Setting $setting)
  {
    $product = $request->product();

    /** @var Money $total */
    $total = $product->price->multiply($request->quantity());

    $feeRate = $setting->fee_rate;

    $feeAmount = $total->multiply($feeRate);
  }
}
```

You can see how we can work with the `Money` object, for example:

```
$total = $product->price->multiply($request->quantity());
```

Because of the cast we discussed earlier, `$product->price` returns a `Money` object that has methods on it. Each of these operations, such as multiplication returns a new `Money` object and doesn't change the original one. These objects are immutable. We have the total amount of the transaction and we also need to calculate the fee:

```
$feeRate = $setting->fee_rate;

$feeAmount = $total->multiply($feeRate);
```

Once again, we use the `multiply` method. The fee rate is stored in a `settings` table that looks like this:

id	fee_rate	foo
1	0.05	bar

It's a "singleton" table, meaning it has only one row. Because of that, we can bind a singleton instance to Laravel's service container:

```
namespace App\Providers;

class AppServiceProvider extends ServiceProvider
{
  public function register(): void
  {
    $this->app->singleton(Setting::class, fn () => Setting::first());
  }
}
```

Whenever you do this:

```
public function store(Setting $setting)
{
  echo $setting->fee_rate;
}
```

Or this:

```
$setting = app(Setting::class);
```

You have the first and only record from the table.

When you have "settings" in your project you can go with this singleton approach, or you can create a key-value style table such as this:

id	key	value
1	FEE_RATE	0.05
2	FOO	bar

Instead of having a single row, you have a row for every setting. I, personally, like the singleton approach better since it's more "reliable." You just look at the table and know exactly what settings your project has. The key-value store is a bit "random." Any developer can insert any value without others knowing about it. Unless you maintain an enum or array with the possible keys. In the singleton model, if you need a new setting, you have to add a new column, with a type, and possibly with a default value as well. It's visible and well-documented. Works better for me. Of course, you can end up with a table that has 50+ columns. Which can be annoying.

So we have a `settings` table. The `fee_rate` column is cast into a `Percent` value object:

```
namespace App\ValueObjects;

class Percent
{
  public string $formattedValue;

  public function __construct(public float $value)
  {
    $this->formattedValue = number_format($this->value * 100, 2) . '%';
  }

  public static function make(float $value): self
  {
    return new static($value);
  }

  /**
   * Percent value is a value such as 15 or 99.93
   */
  public static function fromPercentValue(float $value): self
  {
    return new static($value / 100);
  }
}
```

You'll see the benefits of this class in a minute.

Calculating VAT

The next step is to calculate the VAT rate based on the customer's IP address. For this, I'll use VatStack. They have an API endpoint that returns the exact VAT rate: `https://api.vatstack.com/v1/quotes`

To integrate VatStack I added a new item to the `config/services.php` array:

```
'vatstack' => [
  'public_key' => env('VATSTACK_PUBLIC_KEY'),
  'secret_key' => env('VATSTACK_SECRET_KEY'),
],
```

I also created a `VatService` that communicates with VatStack:

```
namespace App\Services\Vat;

class VatService
{
  public function __construct(private readonly string $publicKey)
  {
  }

  public function calculateVat(Money $amount, string $ipAddress): Vat
  {
    $data = Http::withHeaders([
        'X-API-KEY' => $this->publicKey,
    ])
      ->post('https://api.vatstack.com/v1/quotes', [
        'amount' => $amount->getAmount(),
        'ip_address' => $ipAddress,
      ])
      ->json();

    return Vat::make(
      rate: $data['vat']['rate'],
      amount: $data['vat']['amount'],
    );
```

```
  }
}
```

It's quite simple, just an HTTP request. In a moment, I'll show you the `Vat` object.

This class needs the VatStack public key to work correctly. I inject the value in a new `VatServiceProvider`:

```
class VatServiceProvider extends ServiceProvider
{
  public function register(): void
  {
    $this->app
      ->when(VatService::class)
      ->needs('$publicKey')
      ->give(config('services.vatstack.public_key'));
  }
}
```

VatStack returns a response such as this:

```
array:15 [
  "abbreviation" => "ANUM"
  "amount" => 10000
  "amount_total" => 12700
  "category" => null
  "country_code" => "HU"
  "country_name" => "Hungary"
  "created" => "2024-01-09T09:04:42.464Z"
  "id" => "659d0c2abb8e84ced7a73a98"
  "integrations" => []
  "ip_address" => "77.243.217.87"
  "local_name" => "Közösségi adószám"
  "member_state" => true
  "updated" => "2024-01-09T09:04:42.464Z"
  "validation" => null
  "vat" => array:6 [
    "abbreviation" => "áfa"
```

```
    "inclusive" => false
    "local_name" => "általános forgalmi adó"
    "rate" => 27
    "rate_type" => "standard"
    "amount" => 2700
  ]
]
```

In the `vat` key there's a `rate` and an `amount` value. That's the important stuff:

- The VAT rate is 27%
- For 10000 (which is $10) it's 2700 ($2.7)

VatStack also uses cent values as you can see. They also use integers to represent the percentage value.

And finally the `Vat` value object:

```
namespace App\ValueObjects;

use Money\Money;

class Vat
{
  public function __construct(public int $rate, public Money $amount)
  {
  }

  public static function make(int $rate, int $amount): self
  {
    return new static(
      rate: Percent::fromPercentValue($rate),
      amount: Money::USD($amount),
    );
  }
}
```

It's pretty simple as all value objects should be. Note that VatStack returns amounts in cent values and percentages in int format.

This is how we can use the `VatService`:

```
$vat = $vatService->calculateVat($total, $request->ip());
```

And it returns a `Vat` object with a `Percent` and a `Money` object:

This is the code we have so far:

```
public function store(StoreTransactionRequest $request, VatService
$vatService, Setting $setting)
{
  $product = $request->product();

  /** @var Money $total */
  $total = $product->price->multiply($request->quantity());

  $vat = $vatService->calculateVat($total, $request->ip());

  $feeRate = $setting->fee_rate;

  $feeAmount = $total->multiply((string) $feeRate->value);
}
```

Everything is prepared, now we can create the actual transaction:

```
$transaction = Transaction::create([
  'product_id' => $product->id,
  'quantity' => $request->quantity(),
  'product_data' => $product->toArray(),
  'user_id' => $product->user_id,
  'stripe_id' => Str::uuid(),
  'revenue' => $total,
  'fee_rate' => $feeRate,
  'fee_amount' => $feeAmount,
  'tax_rate' => $vat->rate,
  'tax_amount' => $vat->amount,
  'balance_earnings' => $total->subtract($vat->amount)->subtract($feeAmount),
  'customer_email' => $request->customerEmail(),
]);
```

Since we're using value object casts in the Model we can directly save a `Money` or `Percent` object such as `$total` or `$feeRate`. As I said earlier, we're going to handle Stripe payments later, so now I'm just saving a random UUID in the `stripe_id` column.

Sending webhooks

Users can set up webhooks, such as "trigger this URL when a new transaction is created." I also use this feature in Paddle. When someone buys something from me, Paddle invokes a DigitalOcean function, that sends a request to ConvertKit's API. ConvertKit is my e-mail service provider (e-mail marketing tool such as MailChimp). So I have the customer's e-mail on my mailing list.

The `webhooks` table looks like this:

id	uuid	user_id	url	event
1	9b0bc33c-84a5-4ed3-b3da-a821712595e7	1	https://mysite.com/api/webhooks	transaction_created
2	9b0bc33e-7567-4ace-ab95-2e80d8a1cda2	1	https://mysite.com/api/webhooks	transaction_failed

When someone buys a product that belongs to user 1 we should send a post request to `https://mysite.com/api/webhooks` with the transaction data. The same goes if the transaction fails.

I'd like to send these requests in a queue job in an async way. This is what it looks like from the Controller:

```
if ($webhook = Webhook::transactionCreated($request->user())) {
  SendWebhookJob::dispatch($webhook, 'transaction_created', [
    'data' => $transaction,
  ]);
}
```

`Webhook::transactionCreated` returns a webhook if the user sets up one for the `transaction_created` event and then we can dispatch a job that needs the whole `$transaction` object, the `$webhook` and the event that triggered the webhook.

Here's the job:

```php
class SendWebhookJob implements ShouldQueue
{
  public int $tries = 3;

  public array $backoff = [30, 60];

  use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

  public function __construct(
    private readonly Webhook $webhook,
    private readonly string $event,
    private readonly array $payload,
  ) {}

  public function handle(): void
  {
    try {
      $response = Http::timeout(5)
        ->post($this->webhook->url, $this->payload['data'])
        ->throw();

      WebhookLog::create([
        'webhook_id' => $this->webhook->id,
        'payload' => $this->payload,
        'response_status' => $response->status(),
        'event' => $this->event,
        'status' => WebhookLogStatus::Succeeded,
      ]);
    } catch (RequestException $ex) {
      WebhookLog::create([
        'webhook_id' => $this->webhook->id,
        'payload' => $this->payload,
        'response_status' => $ex->response->status(),
        'error' => $ex->getMessage(),
        'event' => $this->event,
        'status' => WebhookLogStatus::Failed,
```

```
            ]);

            throw $ex;
        } catch (Throwable $ex) {
            WebhookLog::create([
                'webhook_id' => $this->webhook->id,
                'payload' => $this->payload,
                'error' => $ex->getMessage(),
                'event' => $this->event,
                'status' => WebhookLogStatus::Failed,
            ]);

            throw $ex;
        }
    }
}
```

A few important things:

- I'd like to log each and every webhook trigger for observability/debugging purposes. If the response is 2xx we log a `WebhookLogStatus::Succeeded` record. If there was a `RequestException` it means we can gain additional info from the response's status code so we can log that as well. It's important to re-throw the exception from a `catch` block so the job itself can fail too. It's important because we're going to retry them later.
- `$tries =3` means we want to run the job three times if it fails. `$backoff = [30, 60]` means that the second try is going to happen 30s after the first one. The third try will happen 60s after the first one. So we try to run this job three times for 90s if something fails. These are quite high backup values because we send requests and users' websites can be down/slow at any given time. And usually, it takes time before everything goes back to normal. Of course, only experience and trial and error can give you the exact numbers that work best for your project.

Lastly, we need to trigger a webhook if the transaction fails. Here's the complete `TransactionController`:

```
try {
  // Calculating VAT, etc ...

  $transaction = Transaction::create([ ... ]);

  try {
    if ($webhook = Webhook::transactionCreated($request->user())) {
      SendWebhookJob::dispatch($webhook, 'transaction_created', [
          'data' => $transaction,
      ]);
    }
  } catch (Throwable) {}

  return response([
    'data' => TransactionResource::make($transaction)
  ], Response::HTTP_CREATED);
} catch (Throwable $ex) {
  if ($webhook = Webhook::transactionFailed($request->user())) {
    SendWebhookJob::dispatch($webhook, 'transaction_failed', [
        'data' => $request->all(),
    ]);
  }

  throw $ex;
}
```

You'll have logs such as these:

id	uuid	webhook_id	payload	response_status	event	error	status	created_at	updated_at
13	9b0d132…	2	{"data": {"id": 1005, "uuid": "9b0d131e-73ec-4ac3…	404	transaction_created	HTTP request returned status co…	failed	2024-01-09 12:49:44	2024-01-09 12:49:44
14	9b0d135…	2	{"data": {"id": 1005, "uuid": "9b0d131e-73ec-4ac3…	404	transaction_created	HTTP request returned status co…	failed	2024-01-09 12:50:16	2024-01-09 12:50:16
15	9b0d136…	41	{"data": {"id": 1006, "uuid": "9b0d135f-d518-4a34…	404	transaction_created	HTTP request returned status co…	failed	2024-01-09 12:50:22	2024-01-09 12:50:22
16	9b0d138…	41	{"data": {"id": 1007, "uuid": "9b0d1389-95ef-4f3f…	200	transaction_created	NULL	succeeded	2024-01-09 12:50:49	2024-01-09 12:50:49
17	9b0d139…	41	{"data": {"id": 1006, "uuid": "9b0d135f-d518-4a34…	200	transaction_created	NULL	succeeded	2024-01-09 12:50:52	2024-01-09 12:50:52

You can see the `Webhook::transactionCreated` and `Webhook::transactionFailed` methods. In this project, I decided to use custom Eloquent builders. They are simple classes that contain queries for a model. They offer a way to make your models smaller and simpler. Let me show you an example.

First, you need to create a class called `WebhookBuilder`:

```
namespace App\Builders;

use Illuminate\Database\Eloquent\Builder;

class WebhookBuilder extends Builder
{
}
```

It extends the base `Eloquent\Builder` class. The next step is to tell the `Webhook` model to use this custom builder class:

```
namespace App\Models;

class Webhook extends Model
{
  public function newEloquentBuilder($query): WebhookBuilder
  {
    return new WebhookBuilder($query);
  }
}
```

And now we can write queries inside the `WebhookBuilder` class, for example:

```
namespace App\Builders;

class WebhookBuilder extends Builder
{
  public function transactionFailed(User $user): ?Webhook
  {
    return $this
      ->where('user_id', $user->id)
      ->where('event', WebhookEvent::TransactionFailed)
      ->first();
  }
}
```

This is a regular Eloquent query but instead of using `self::where` or `Webhook::where` we can just write `$this->where`.

Now whenever you write something like `Webhook::transactionFailed` Laravel will look for a method called `transactionFailed` in the `WebhookBuilder` class. If you want to learn more about custom builders read this article.

Here's the whole class:

```
namespace App\Builders;

class WebhookBuilder extends Builder
{
  public function transactionFailed(User $user): ?Webhook
  {
    return $this->for($user, WebhookEvent::TransactionFailed);
  }

  public function transactionCreated(User $user): ?Webhook
  {
    return $this->for($user, WebhookEvent::TransactionCreated);
  }

  public function payoutSent(User $user): ?Webhook
  {
```

```
    return $this->for($user, WebhookEvent::PayoutSent);
  }

  private function for(User $user, WebhookEvent $event): ?Webhook
  {
    return $this
      ->where('user_id', $user->id)
      ->where('event', $event->value)
      ->first();
  }
}
```

`WebhookEvent` is a simple, string backed enum:

```
namespace App\Enums;

enum WebhookEvent: string
{
  case TransactionCreated = 'transaction_created';
  case TransactionFailed = 'transaction_failed';
  case PayoutSent = 'payout_sent';
}
```

By using custom builders you can have really simple models. After all, a model should be a representation of a database record. Something like this:

```
namespace App\Models;

class Webhook extends Model
{
  use HasFactory;

  protected $fillable = [
    'user_id',
    'url',
    'event',
  ];

  public function logs(): HasMany
  {
    return $this->hasMany(WebhookLog::class);
  }

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

Exporting transactions

For the next feature, let's export transactions in users' favorite format: Excel.

For this, I'm going to use the excellent laravel-excel package. We have a number of options but the main factor for now is performance. Users can have tens of thousands or even millions of transactions. The most basic would be something like this:

```
namespace App\Exports;

class TransactionExport implements FromCollection
{
  public function __construct(private readonly User $user)
  {
  }

  public function collection()
  {
    return Transaction::where('user_id', $this->user->id);
  }
}
```

This is a working export that showcases the excellence of this package. However, this has a small problem. Let's say the given user has 500 000 transactions:

- The package internally calls the `collection` method
- The query returns with 500k transactions. An average `Transaction` object is about 3KB. 500k transactions is about 1.5 million KB which is 1464MB. If ten users click on `Export` at the same time we are using almost 15GB of RAM.
- Then the package loops through the rows and appends them to an XLS. I don't know the internals of laravel-excel but I guess it's even more memory usage.

Another potential and pretty common problem with collection-based export is that it's easy to make N+1 query problems. Let's say you want to include the username based on the transaction's user relationship. You can do it like this:

```
public function map($transaction): array
{
  return [
     ...
    $transaction->user->username,
  ];
}
```

You just made an additional 500,000 database queries.

Let me repeat that:

- This export runs 500k queries
- And uses at least 1.5GB of RAM

Now, anytime a user with 500k transactions clicks on the `Export` button you'll have a Slack alert about some downtime or slow response times.

Let's not make these mistakes!

Instead of a Collection, we can use a query as well:

```
namespace App\Exports;

class TransactionsExport implements FromQuery
{
  public function __construct(
    private readonly User $user,
    private readonly DateInterval $interval,
  ) {}

  public function query()
  {
    return Transaction::query()
      ->select([
        'uuid',
        'product_data',
        'quantity',
        'revenue',
        'fee_amount',
        'tax_amount',
        'balance_earnings',
        'customer_email',
        'created_at',
      ])
      ->where('user_id', $this->user->id)
      ->whereBetween('created_at', [
        $this->interval->startDate->date,
        $this->interval->endDate->date,
      ])
      ->orderBy('created_at');
  }
}
```

Don't worry about the `DateInterval` for now.

Behind the scenes, this query will be executed in chunks. So instead of querying and loading 500k records into the memory, laravel-excel will query and load only a small chunk at a time.

You can even customize the chunk size by implementing the `WithCustomChunkSize` interface:

```
class TransactionsExport implements FromQuery, WithCustomChunkSize
{
  public function query()
  {
    //  ...
  }

  public function chunkSize(): int
  {
    return 250;
  }
}
```

Now laravel-excel will query and handle 250 records at once. Unfortunately, no magic number works best for every project. Test your exports with different settings and decide based on the results. Also, note that chunking is no magic pill. It saves memory, of course, but usually, it uses more CPU since it needs to do more things.

The export is still not ready. In the DB we store money values in cents. So if you run this export it will show numbers such as 2131 which in reality would be $21,31. Fortunately, laravel-excel comes with mappings:

```
namespace App\Exports;

class TransactionsExport implements FromQuery, WithCustomChunkSize,
WithMapping
{
  public function query() {}
  public function chunkSize(): int {}

  public function map($row): array
  {
    return [
      $row->uuid,
      Arr::get($row->product_data, 'title'),
      $row->quantity,
      MoneyForHuman::from($row->revenue)->value,
      MoneyForHuman::from($row->fee_amount)->value,
      MoneyForHuman::from($row->tax_amount)->value,
      MoneyForHuman::from($row->balance_earnings)->value,
      $row->customer_email,
      $row->created_at,
    ];
  }
}
```

In the `map` method we can return an array that contains the formatted values in the same order as the `select()` in the `query` function.

You can see a new value object called `MoneyForHuman` that takes a `Money` object and formats it in a human-readable format, such as $29.99

```php
<?php

namespace App\ValueObjects;

use Money\Currencies\ISOCurrencies;
use Money\Formatter\IntlMoneyFormatter;
use Money\Money;
use NumberFormatter;

class MoneyForHuman
{
  public readonly string $value;

  public function __construct(private readonly Money $money)
  {
    $currencies = new ISOCurrencies();

    $numberFormatter = new NumberFormatter(
      'en_US',
      NumberFormatter::CURRENCY
    );

    $moneyFormatter = new IntlMoneyFormatter($numberFormatter, $currencies);

    $this->value = $moneyFormatter->format($this->money);
  }

  public static function from(Money $money): self
  {
    return new self($money);
  }

  public function __toString(): string
  {
    return $this->value;
  }
```

```
}
```

This way, anytime you need to show a dollar amount you just do `MoneyForHuman($transaction->revenue)` for example. Another solution would be to create a class called `Money` that extends the base `Money\Money` class and has a method `getHumanAmount()` or something like that.

In the final XLSX file, we want a user-friendly header row. We can use the `WithHeadings` interface:

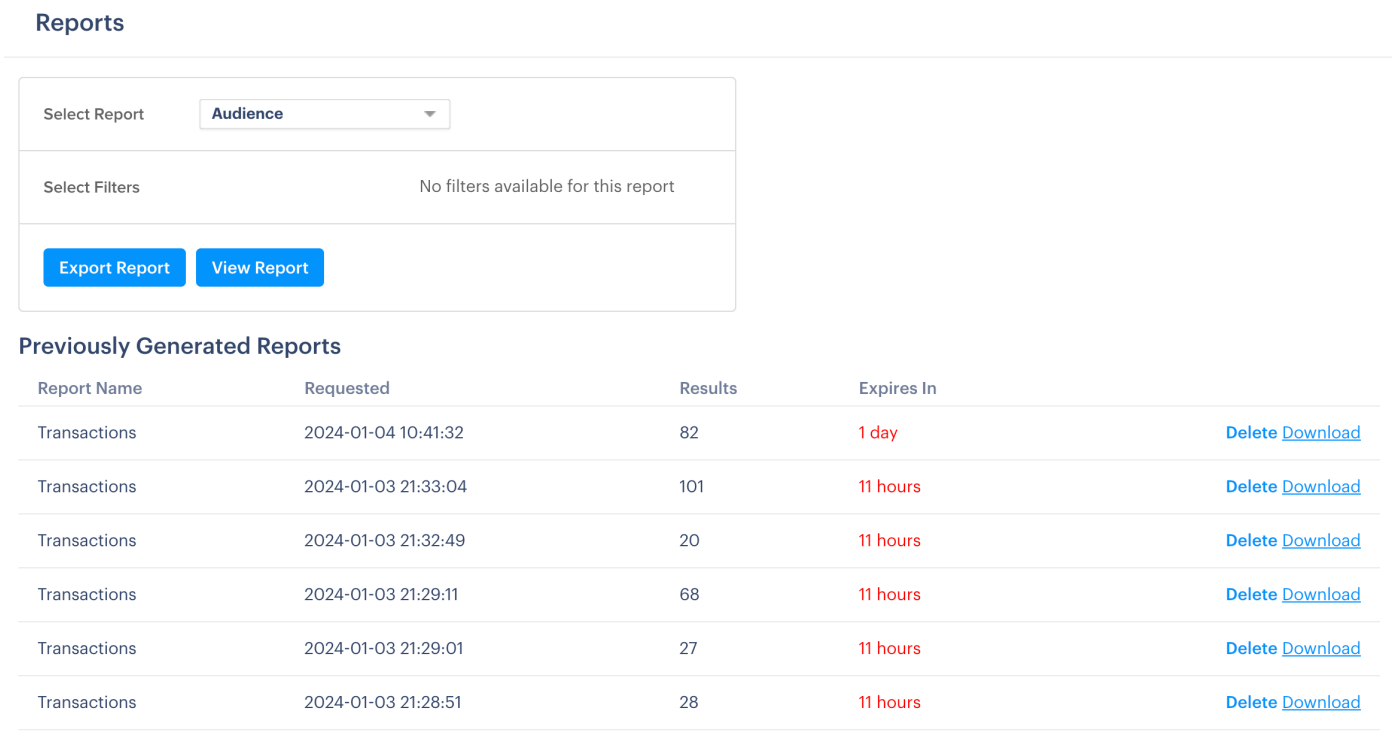
```
class TransactionsExport implements FromQuery, WithCustomChunkSize,
WithMapping, WithHeadings
{
  public function query() {}
  public function chunkSize(): int {}
  public function map($row): array {}

  public function headings(): array
  {
    return [
      '#',
      'Product',
      'Quantity',
      'Total',
      'Fee',
      'Tax',
      'Balance earnings',
      'Customer e-mail',
      'Date',
    ];
  }
}
```

And that's it (for now). This basic export results in documents such as this one:

#	Product	Quantity	Total	Fee	Tax	Balance earnings	Customer e-mail	Date
9b0bc33b-dd48-4095-a964-f2af93082a4a	labore magni explicabo	2	$73.63	$3.68	$11.04	$58.91	kuhn.claude@bruen.biz	2023-12-01 21:10:24
9b0bc33b-dbba-431b-90b2-227a046de617	labore magni explicabo	2	$98.14	$4.91	$14.72	$78.51	kilback.maia@kub.biz	2023-12-03 21:10:24
9b0bc33b-de36-49d2-8cec-8842e0b2e650	labore magni explicabo	8	$96.90	$4.85	$14.54	$77.51	xruecker@hotmail.com	2023-12-04 21:10:24
9b0bc33b-c0ff-4c31-8699-e33273e72178	labore magni explicabo	9	$10.34	$0.52	$1.55	$8.27	bhane@gmail.com	2023-12-06 21:10:24
9b0bc33b-c846-4d48-81d2-c280089ef877	labore magni explicabo	2	$51.69	$2.58	$7.75	$41.36	emmanuelle77@franecki.com	2023-12-08 21:10:24
9b0bc33b-c9e8-41bc-83c9-249b0dbc1af2	labore magni explicabo	5	$34.23	$1.71	$5.13	$27.39	kharvey@hudson.org	2023-12-09 21:10:24
9b0bc33b-d3da-4d9f-b26a-83bf5db15083	labore magni explicabo	9	$79.18	$3.96	$11.88	$63.34	block.allene@hotmail.com	2023-12-09 21:10:24
9b0bc33b-d541-4e62-8db8-b9c4f4e617fd	labore magni explicabo	10	$10.22	$0.51	$1.53	$8.18	king.raheem@west.com	2023-12-09 21:10:24
9b0bc33b-c18c-421e-9960-179e8e6dd226	labore magni explicabo	5	$26.54	$1.33	$3.98	$21.23	gabe.torphy@bartoletti.com	2023-12-10 21:10:24
9b0bc33b-d4ce-416f-9158-95a22f721e64	labore magni explicabo	6	$42.13	$2.11	$6.32	$33.70	steuber.jailyn@mcglynn.com	2023-12-11 21:10:24
9b0bc33b-ddbf-4452-99f4-8b5f42f70a37	labore magni explicabo	10	$45.38	$2.27	$6.81	$36.30	santino.green@yahoo.com	2023-12-11 21:10:24

Transaction reports

Paddle preserves your reports for a week and then they delete them. It's necessary to save disk space. On the reports page, they list the available reports from the past week:



In order to do that we need a model called `TransactionReport` that looks like this:

id	uuid	user_id	number_of_transactions	status	expired_at
1	9b0de58a-8a9a-4b03-9371-ee891c4c3302	1	82	done	2024-01-11 13:52:01
2	9b0de379-cd3d-402b-8ae9-d80abe7d9547	1	101	done	2024-01-10 22:57:19
3	9b0de4cb-9589-46e7-8bc5-31a74ff7ed40	1	19	pending	2024-01-17 15:34:51

Before running the actual Export class we need to create a `TransactionReport` object with a `pending` status and if the export succeeds it can be marked as `done` otherwise as `failed`.

The API endpoint is simply: `POST /api/transaction-reports` and the Controller looks like this:

```php
namespace App\Http\Controllers;

class TransactionReportController extends Controller
{
  public function store(StoreTransactionReportRequest $request)
  {
    /** @var TransactionReport $report */
    $report = TransactionReport::create([
      'user_id' => $request->user()->id,
      'expires_at' => now()->addWeek(),
      'status' => TransactionReportStatus::Pending,
    ]);

    ExportTransactionsAction::execute(
      $request->user(),
      $request->interval(),
      $report,
    );

    return response([
      'data' => 'Your data is being exported',
    ], Response::HTTP_ACCEPTED);
  }
}
```

First, we create the `TransactionReport` object in the `pending` state. The default expiration is one week. Then I created an action called `ExportTransactionAction` that triggers the actual `TransactionsExport` as a queue job:

```php
namespace App\Actions;

class ExportTransactionsAction
{
  public static function execute(
    User $user,
    DateInterval $interval,
    TransactionReport $report,
  ): void {
    $export = new TransactionsExport(
      $user,
      $interval,
      $report,
    );

    $report->number_of_transactions = $export->query()->count();

    $report->save();

    $export->queue($report->relativePath());
  }
}
```

It creates the export object, queries the count of transactions, saves it to the `TransactionReport`, and then it dispatches the job.

The reason we need this action is that later we need to run the export when paying out users. Each payout has its own report. So this piece of code needs to be reusable, this is why it has its own action.

Finally, the `StoreTransactionReportRequest`:

```php
namespace App\Http\Requests;

class StoreTransactionReportRequest extends FormRequest
{
  public function rules(): array
  {
    return [
      'start_date' => ['required', 'date'],
      'end_date' => ['required', 'date'],
    ];
  }

  public function interval(): DateInterval
  {
    return DateInterval::make(
      StartDate::fromString($this->start_date),
      EndDate::fromString($this->end_date),
    );
  }
}
```

The endpoint accepts `start_date` and `end_date` parameters but it creates a `DateInterval` value object from these values and the whole application works with this object instead of string values. The `DateInterval` is really simple:

```php
namespace App\ValueObjects;

class DateInterval
{
  public function __construct(
    public readonly StartDate $startDate,
    public readonly EndDate $endDate,
  ) {}

  public static function make(StartDate $startDate, EndDate $endDate): self
  {
```

```
        return new self($startDate, $endDate);
    }
}
```

It's just a container with a `StartDate` and `EndDate` objects.

`StartDate` is an object that **always** represents the start of a day:

```
namespace App\ValueObjects;

class StartDate
{
    public Carbon $date;

    public function __construct(Carbon $date)
    {
        $this->date = $date->startOfDay();
    }

    public static function fromString(string $date): self
    {
        return new static(Carbon::parse($date));
    }

    public static function fromCarbon(Carbon $date): self
    {
        return new static($date);
    }

    public function __toString(): string
    {
        return $this->date->format('Y-m-d H:i:s');
    }
}
```

By doing this in the constructor: `$this->date = $date->startOfDay();` we are enforcing the rule that "when users want to query/export reports it always starts at 00:00:00" This way, we are encapsulating this business rule in a domain-specific value object.

The same goes for `EndDate`:

```
namespace App\ValueObjects;

class EndDate
{
  public Carbon $date;

  public function __construct(Carbon $date)
  {
    $this->date = $date->endOfDay();
  }

  public static function fromString(string $date): self
  {
    return new static(Carbon::parse($date));
  }

  public static function fromCarbon(Carbon $date): self
  {
    return new static($date);
  }

  public function __toString(): string
  {
    return $this->date->format('Y-m-d H:i:s');
  }
}
```

Now that we have a `TransactionReport` object before dispatching the job, the last thing is to update its state based on the job's status:

```
class TransactionsExport implements FromQuery, WithHeadings, WithEvents,
ShouldAutoSize, WithCustomChunkSize, WithMapping
{
  public function failed(Throwable $exception): void
  {
    $this->report->status = TransactionReportStatus::Failed;

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

  public function registerEvents(): array
  {
    return [
      AfterSheet::class => function () {
        $this->report->status = TransactionReportStatus::Done;

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

When something goes wrong the `failed` method is called so we can mark the report as `Failed`. The `AfterSheet` event is raised when laravel-excel is done with the sheet so we can mark the report as `Done`.

Notifying the user

Since exporting a report is an async operation, in the Controller we respond with a `202 Accepted` status code:

```
return response([
  'data' => 'Your data is being exported',
], Response::HTTP_ACCEPTED);
```

When the export is done we have to send an e-mail to the user just as Paddle does.

In the action, where we have this line:

```
$export->queue($report->relativePath());
```

We need to chain another job after the export job is finished. Fortunately, laravel-excel provides us a method called `chain` that works such as this:

```
$export->queue($report->relativePath())
  ->chain([
    new Job1(),
    new Job2(),
  ]);
```

This will run these jobs in order:

- Export
- Job1
- Job2

If one of the jobs fails, the chain fails too. This feature relies on Laravel's job chaining.

So all we need is a new job that sends a notification:

```
$export
  ->queue($report->relativePath())
  ->chain([
      new NotifyUserAboutExportJob($user, $report),
  ]);
```

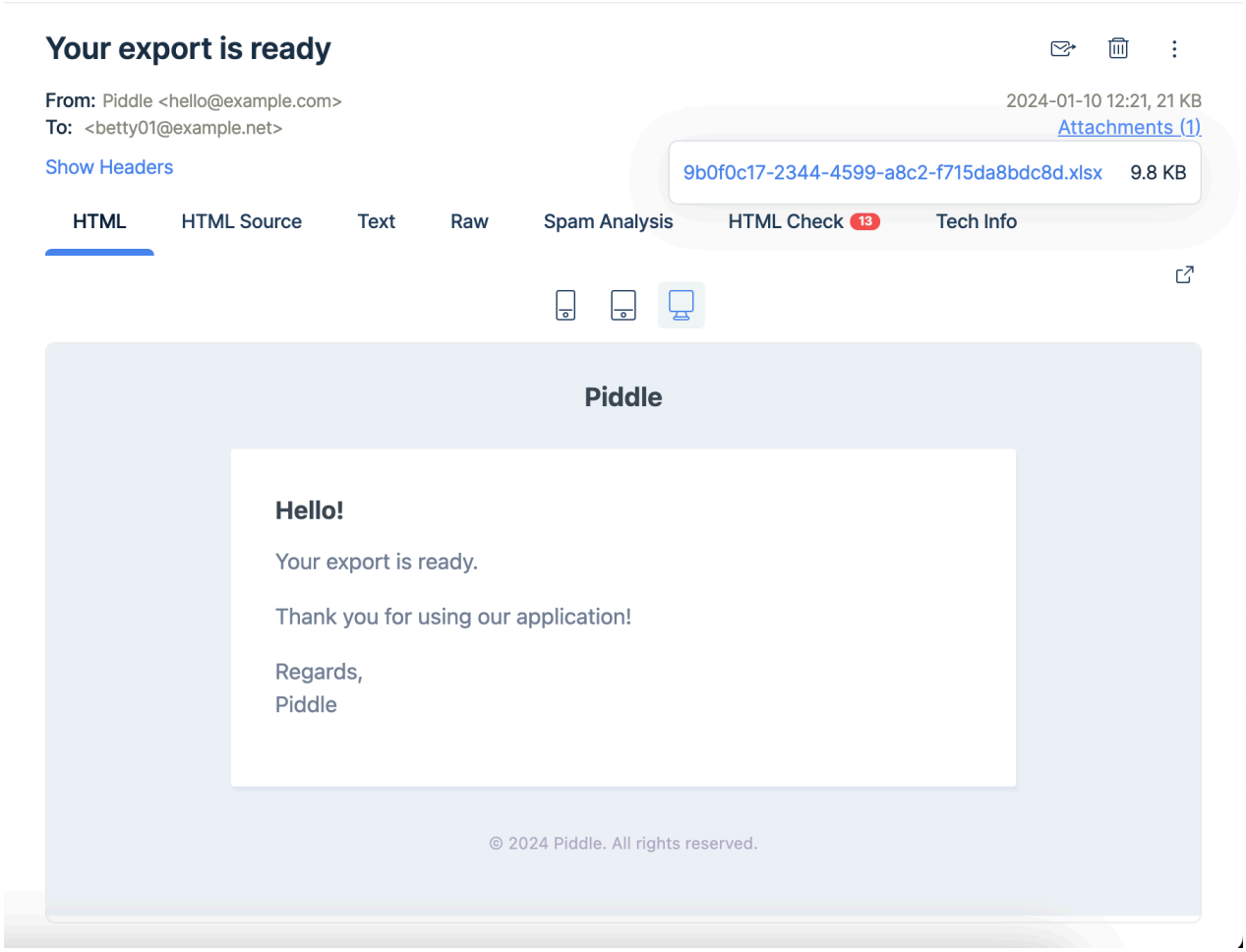
The job itself is pretty straightforward:

```
namespace App\Jobs;

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

  public function __construct(
    private readonly User $user,
    private readonly TransactionReport $transactionReport,
  ) {}

  public function handle(): void
  {
    $this->user->notify(
      new ExportReadyNotification(
        storage_path($this->transactionReport->absolutePath()),
      ),
    );
  }
}
```

It just calls `$this->user->notify` with an `ExportReadyNotification` that implements a mail notification:

```php
namespace App\Notifications;

class ExportReadyNotification extends Notification implements ShouldQueue
{
  use Queueable;

  public function __construct(private readonly string $storagePath)
  {
  }

  public function via(object $notifiable): array
  {
    return ['mail'];
  }

  public function toMail(object $notifiable): MailMessage
  {
    return (new MailMessage)
      ->line('Your export is ready.')
      ->line('Thank you for using our application!')
      ->attach($this->storagePath);
  }
}
```

It has the XLSX as an attachment. If you use Mailtrap it's pretty simple to test the mail:


Your export is ready
From: Piddle <hello@example.com>
To: <betty01@example.net>
2024-01-10 12:21, 21 KB
Attachments (1)
9b0f0c17-2344-4599-a8c2-f715da8bdc8d.xlsx 9.8 KB
Show Headers
HTML
HTML Source
Text
Raw
Spam Analysis
HTML Check 13
Tech Info
Piddle
Hello!
Your export is ready.
Thank you for using our application!
Regards,
Piddle
© 2024 Piddle. All rights reserved.

Cleaning up expired transaction reports

The last thing that needs to be done is cleaning up expired reports and files. This is pretty easy:

```
namespace App\Jobs;

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

  public function handle(): void
  {
    $reports = TransactionReport::query()
      ->where('expires_at', '≤', now())
      ->get();

    foreach ($reports as $report) {
      /** @var TransactionReport $report */
      Storage::delete($report->relativePath());

      $report->delete();
    }
  }
}
```

I added a small helper method to the `TransactionReport` model:

```
public function relativePath(): string
{
  return "exports/{$this->uuid}.xlsx";
}
```

Even if I use dedicated query builder classes I still write these small getter-type methods on the Model. By the way, if you have the opportunity, don't include user-related information in file names such as usernames, full names, company names, and stuff like that. Use something like a UUID.

And of course, we need to schedule the job:

```
namespace App\Console;

class Kernel extends ConsoleKernel
{
  protected function schedule(Schedule $schedule): void
  {
    $schedule->job(CleanUpExpiredTransactionReportsJob::class)->daily();
  }
}
```

Sending payouts

The next big feature is paying out users. This happens once every month, and we pay them the sum of last month's transactions. Here's the plan:

- Get users
- Reject the ones who already received payment this month
- Get last month's transactions
- Generate a transaction report
- Sum up the earnings
- Make a Stripe transfer
- Create a payout object
- Trigger a webhook

Here's the first part:

```
namespace App\Jobs;

class SendPayoutsJob implements ShouldQueue
{
  public function handle(DateService $dateService): void
  {
    foreach (User::all() as $user) {
      /** @var User $user */
      if (!$user->shouldReceivePayout($dateService->lastMonth())) {
          return;
      }

      /** @var TransactionReport $report */
      $report = TransactionReport::create([
          'user_id' => $user->id,
          'expires_at' => $dateService->infinity(),
          'status' => TransactionReportStatus::Pending,
      ]);

      $payoutUuid = Str::uuid();
    }
  }
}
```

First, we need to reject users who already received a payment this month. Let's say you dispatch this job, but there were some errors with 3 users. You should be able to re-run it without any side effects. This check avoids paying the same user twice.

I created a small "helper" class called `DateService`:

```
namespace App\Services;

class DateService
{
  public static function lastMonth(): DateInterval
  {
    return DateInterval::make(
      startDate: StartDate::fromCarbon(now()->subMonthNoOverflow()->startOfMonth()),
      endDate: EndDate::fromCarbon(now()->subMonthNoOverflow()->endOfMonth()),
    );
  }

  public static function thisMonth(): DateInterval
  {
    return DateInterval::make(
      startDate: StartDate::fromCarbon(now()->startOfMonth()),
      endDate: EndDate::fromCarbon(now()->endOfMonth()),
    );
  }
}
```

The responsibility of this class is to encapsulate and generalize some basic date-related functions. Later, we'll extend it with more methods.

Then in the `UserBuilder` we need a `shouldReceivePayout` method:

```
namespace App\Builders;

class UserBuilder extends Builder
{
  public function shouldReceivePayout(DateInterval $interval): bool
  {
    $hasTransactions = $this->model->transactions()
      ->whereBetween('created_at', [
        $interval->startDate->date,
        $interval->endDate->date,
      ])
      ->exists();

    if (!$hasTransactions) {
      return false;
    }

    $alreadyReceivedPayout = $this->model->payouts()
      ->where('year', $interval->startDate->date->year)
      ->where('month', $interval->startDate->date->month)
      ->where('status', '≠', PayoutStatus::Failed)
      ->exists();

    return !$alreadyReceivedPayout;
  }
}
```

A few important things:

- In a `Builder` class you can access the current model by using the `$this->model` property. In this case, it's a `User` instance.
- If the user does not have transactions at all we don't need to send a payout.
- If the user already received a successful payment they shouldn't get another one.

You haven't seen the `Payout` model so far. This is what it looks like:

id	uuid	user_id	amount	year	month	paid_at	status	stripe_id	txn_report_id
1	9b0bc33b-e6a7-48f1-b4bd-7a8ca0908aff	1	12000	2024	1	2024-01-10 14:22:00	succeeded	asdf-1234	1
2	9b0bc33c-1586-4b97-a034-5096ffba44bc	2	9900	2024	1	NULL	pending	NULL	2

Now we have another three things to do:

- Exporting a report
- Sending the payment
- Notifying the user

All of them are async queue jobs. We need to use a `chain` to execute them in order. It's similar to what we did earlier with laravel-excel but now it's pure Laravel:

```
Bus::chain([
  new Job1,
  new Job2,
  fn () => var_dump('hey'),
])
```

You can use jobs or closures as well:

```
Bus::chain([
  function () use ($user, $report, $dateService) {
    try {
      ExportTransactionsAction::execute(
        $user,
        $dateService->lastMonth(),
        $report,
      );
    } catch (Exception) {
      $report->status = TransactionReportStatus::Failed;

      $report->save();
    }
  },
  new SendPayoutJob($user, $report, $payoutUuid, $dateService->lastMonth()),
  new SendPayoutNotificationJob($user, $payoutUuid, $report),
])->dispatch();
```

First, I used a Closure to export the report for the following reason: I don't want the chain to fail if the report cannot be exported. Even though, the user won't be able to download an XLSX immediately, it's more important to silence this exception and try to send them the money, in my opinion. So if the action fails, I just set the status to `Failed` and move on.

The next part is the `SendPayoutJob`:

```
namespace App\Jobs;

class SendPayoutJob implements ShouldQueue
{
  public function __construct(
    private readonly User $user,
    private readonly DateInterval $period,
  ) {}

  public function handle(StripeClient $stripe): void
  {
    $query = $this->user
      ->transactions()
      ->whereBetween('created_at', [
        $this->period->startDate->date,
        $this->period->endDate->date,
      ]);

    $amount = Money::USD($query->sum('balance_earnings'));
  }
}
```

With this simple query, we have the amount that the user must receive.

Stripe

This is not going to be a Stripe tutorial by any means. They have an awesome documentation and there are countless Youtube videos.

The next step is to actually send money. For this, I'm going to use Stripe. There's a pretty good SDK called `stripe/stripe-php`. Let's add an API key:

```
STRIPE_API_KEY=sk_test_blah
```

I like to add all of these 3rd party service into the `config/services.php` config:

```
'stripe' => [
  'key' => env('STRIPE_API_KEY'),
],
```

And I also created a new provider called `StripeServiceProvider` that provides the API key to `stripe/stripe-php`'s `StripeClient` class:

```
namespace App\Providers;

class StripeServiceProvider extends ServiceProvider
{
  public function register(): void
  {
    $this->app->singleton(
      StripeClient::class,
      fn () => new StripeClient(
        config('services.stripe.key')
      ),
    );
  }
}
```

The next step is to send the money:

```
public function handle(StripeClient $stripe): void
{
  //  ...

  $amount = Money::USD($query->sum('balance_earnings'));

  try {
    $stripeTransfer = $stripe->transfers->create([
      'amount' => $amount->getAmount(),
      'currency' => 'USD',
      'destination' => $this->user->stripe_account_id,
    ]);
  } catch (Throwable $e) {
    Payout::create([
        'status' => PayoutStatus::Failed,
        'paid_at' => null,
        'stripe_id' => null,
        'uuid' => $this->uuid,
        'user_id' => $this->user->id,
        'year' => $this->period->startDate->date->year,
        'month' => $this->period->startDate->date->month,
        'amount' => $amount,
        'transaction_report_id' => $this->report->id,
    ]);

    throw $e;
  }
}
```

From Stripe docs: *A `Transfer` object is created when you move funds between Stripe accounts as part of Connect.*

The idea is that users set up their Stripe account and we store the ID in the `stripe_account_id` column so we can move funds using the `Transfer` API.

That's it. If the transfer fails we create a `Payout` with a status of `Failed` so we can retry it later. As I said earlier, this job is designed in a way, that you can run it anytime without causing side effects.

If Stripe transfer is successful we can create a successful Payout object:

```
$payout = Payout::create([
  'uuid' => $this->uuid,
  'user_id' => $this->user->id,
  'paid_at' => now(),
  'year' => $this->period->startDate->date->year,
  'month' => $this->period->startDate->date->month,
  'amount' => $amount,
  'status' => PayoutStatus::Succeeded,
  'stripe_id' => $stripeTransfer->id,
  'transaction_report_id' => $this->report->id,
]);
```

Yes, I duplicated this block of code and changed only a few lines. It looks like this:

```
try {
  $stripeTransfer = $stripe->transfers->create([
    'amount' => $amount->getAmount(),
    'currency' => 'USD',
    'destination' => $this->user->stripe_account_id,
  ]);
} catch (Throwable $e) {
  Payout::create([
    'status' => PayoutStatus::Failed,
    'paid_at' => null,
    'stripe_id' => null,
    'uuid' => $this->uuid,
    'user_id' => $this->user->id,
    'year' => $this->period->startDate->date->year,
    'month' => $this->period->startDate->date->month,
    'amount' => $amount,
    'transaction_report_id' => $this->report->id,
  ]);

  throw $e;
}
```

```
$payout = Payout::create([
  'uuid' => $this->uuid,
  'user_id' => $this->user->id,
  'paid_at' => now(),
  'year' => $this->period->startDate->date->year,
  'month' => $this->period->startDate->date->month,
  'amount' => $amount,
  'status' => PayoutStatus::Succeeded,
  'stripe_id' => $stripeTransfer->id,
  'transaction_report_id' => $this->report->id,
]);
```

Another solution would be this:

```
$payout = Payout::create([
  'status' => PayoutStatus::Pending,
  'paid_at' => null,
  'stripe_id' => null,
  'uuid' => $this->uuid,
  'user_id' => $this->user->id,
  'year' => $this->period->startDate->date->year,
  'month' => $this->period->startDate->date->month,
  'amount' => $amount,
  'transaction_report_id' => $this->report->id,
]);

try {
  $stripeTransfer = $stripe->transfers->create([
    'amount' => $amount->getAmount(),
    'currency' => 'USD',
    'destination' => $this->user->stripe_account_id,
  ]);
} catch (Throwable $e) {
  $payout->status = PayoutStatus::Failed;

  $payout->save();
```

```
    throw $e;
}

$payout->status = PayoutStatus::Succeeded;
$payout->paid_at = now();
$payout->stripe_id = $stripeTransfer->id;

$payout->save();
```

I know it's highly subjective but I prefer the first one with code duplication. Please pardon my outrageous behavior.

The next step is to update all the transactions and fill out the `payout_id` column:

```
$transactionIds = $query->select('id')->pluck('id');

Transaction::query()
    ->whereIn('id', $transactionIds)
    ->update(['payout_id' => $payout->id]);
```

And finally, we can trigger a webhook:

```
if ($webhook = Webhook::payoutSent($this->user)) {
    SendWebhookJob::dispatch($webhook, 'payout_sent', [
        'data' => $payout,
    ]);
}
```

This is the same as before only the payload is different.

In this method, we are creating a new Payout record and potentially updating thousands of transaction records, and also communicating with Stripe, and moving funds. Lots of things can go wrong. This is why it's a good idea to put everything into one MySQL transaction. So the final method looks like this:

```php
public function handle(StripeClient $stripe): void
{
  DB::transaction(function () use ($stripe) {
    $query = $this->user
      ->transactions()
      ->whereBetween('created_at', [
        $this->period->startDate->date,
        $this->period->endDate->date,
      ]);

    $amount = Money::USD($query->sum('balance_earnings'));

    try {
      $stripeTransfer = $stripe->transfers->create([
        'amount' => $amount->getAmount(),
        'currency' => 'USD',
        'destination' => $this->user->stripe_account_id,
      ]);
    } catch (Throwable $e) {
      Payout::create([
        'status' => PayoutStatus::Failed,
        'paid_at' => null,
        'stripe_id' => null,
        'uuid' => $this->uuid,
        'user_id' => $this->user->id,
        'year' => $this->period->startDate->date->year,
        'month' => $this->period->startDate->date->month,
        'amount' => $amount,
        'transaction_report_id' => $this->report->id,
      ]);

      throw $e;
    }

    $payout = Payout::create([
      'uuid' => $this->uuid,
```

```
      'user_id' => $this->user->id,
      'paid_at' => now(),
      'year' => $this->period->startDate->date->year,
      'month' => $this->period->startDate->date->month,
      'amount' => $amount,
      'status' => PayoutStatus::Succeeded,
      'stripe_id' => $stripeTransfer->id,
      'transaction_report_id' => $this->report->id,
    ]);

    $transactionIds = $query->select('id')->pluck('id');

    Transaction::query()
      ->whereIn('id', $transactionIds)
      ->update(['payout_id' => $payout->id]);

    if ($webhook = Webhook::payoutSent($this->user)) {
      SendWebhookJob::dispatch($webhook, 'payout_sent', [
        'data' => $payout,
      ]);
    }
  });
}
```

Queues and workers

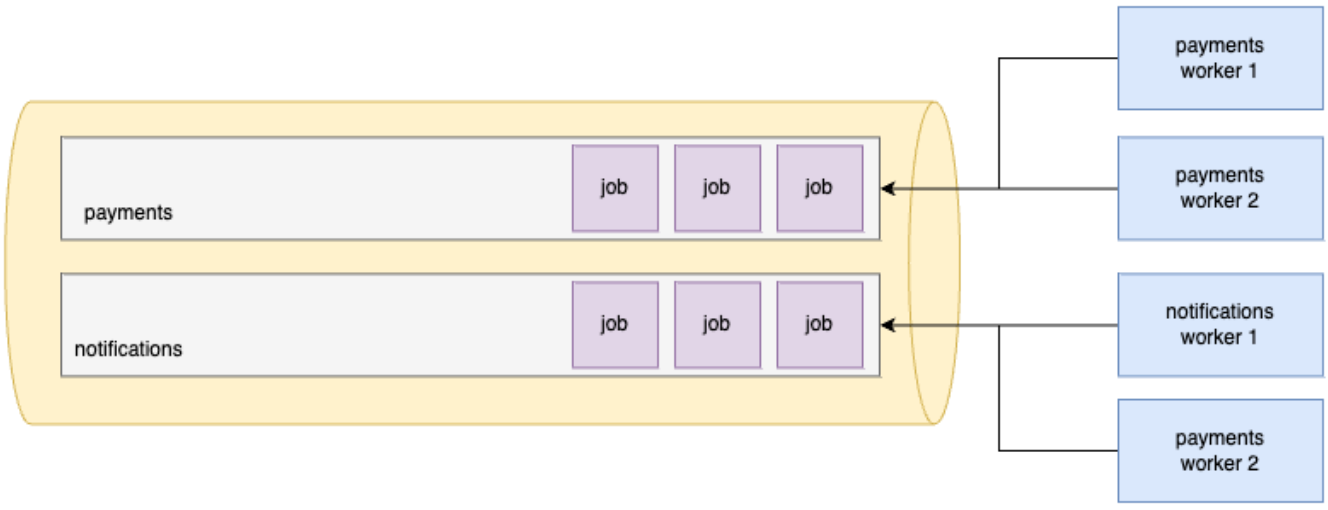
There's only one thing left to do. If remember I said in the introduction that if you use only one queue for everything you'll have some problematic situations. I'm not going to repeat it, but imagine there are 10,000 notifications in the queue, and you dispatch the payout jobs.

To avoid these situations all you need to do is to use multiple queues. In this project, we want to have a dedicated `payout` queue that has only payout jobs. Basically, we only need to instruct Laravel that it should dispatch the payout jobs on a `payout` queue:

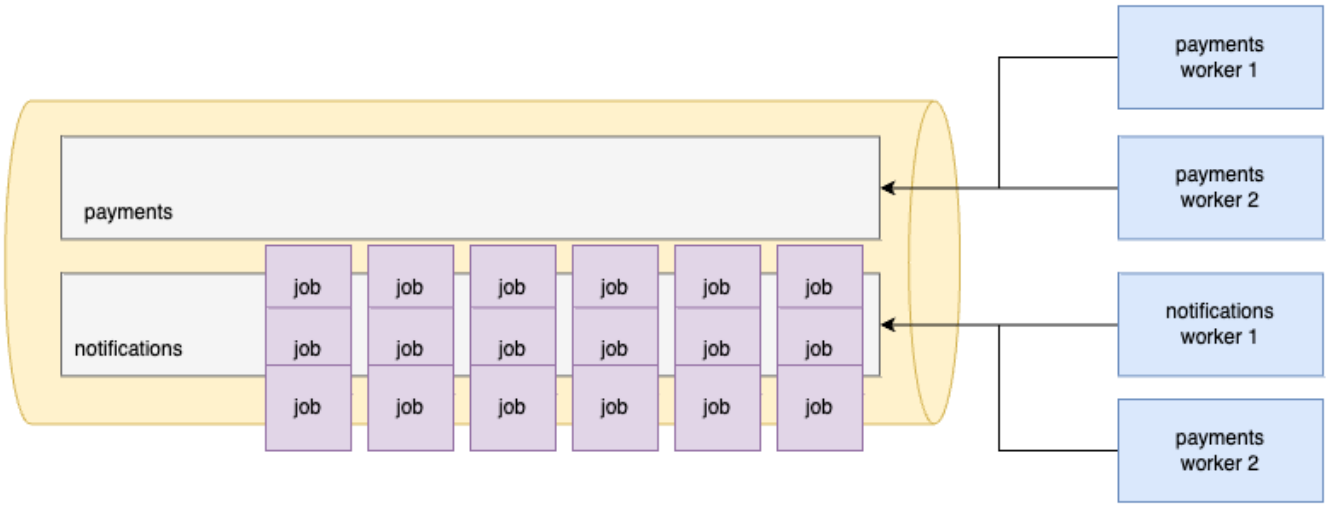
```
class SendPayoutJob implements ShouldQueue
{
  public function __construct(
      private readonly User $user,
      private readonly TransactionReport $report,
      private readonly string $uuid,
      private readonly DateInterval $period,
  ) {
      $this->onQueue('payout');
  }
}
```

`$this->onQueue('payout')` ensures that this job is always dispatched onto the `payout` queue.

The next step is to start a worker that processes the `payout` queue. There's a `--queue` flag in the `queue:work` command so we can do this: `php artisan queue:work --queue=payout` And there can be another worker process that processes the default queue: `php artisan queue:work --queue=default` You could think that it's a good idea to have a dedicated worker only for the payout queue, right? After all, these are the most important jobs. But actually, it's not true. I'll show you why.



In this example, there are two queues, `payout` and `notifications`. The two are being processed at the same time by dedicated workers. Which is great, but what if something like that happens?



There are so many jobs in the notifications queue but none in the payout. If that happens we just waste all the payout worker processes since they have nothing to do. But this command doesn't let them process anything else:

```
php artisan queue:work --queue=payout
```

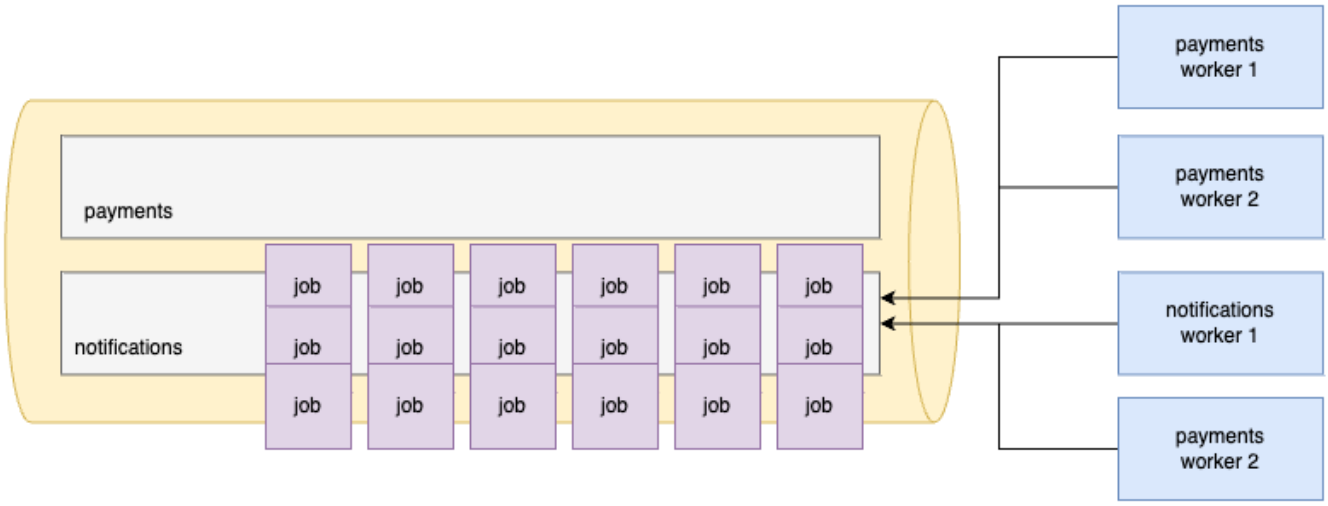
This means they can **only** touch the payout queue and nothing else.

Because of that problem, I don't recommend you to have dedicated workers for only one queue. Instead, prioritize them!

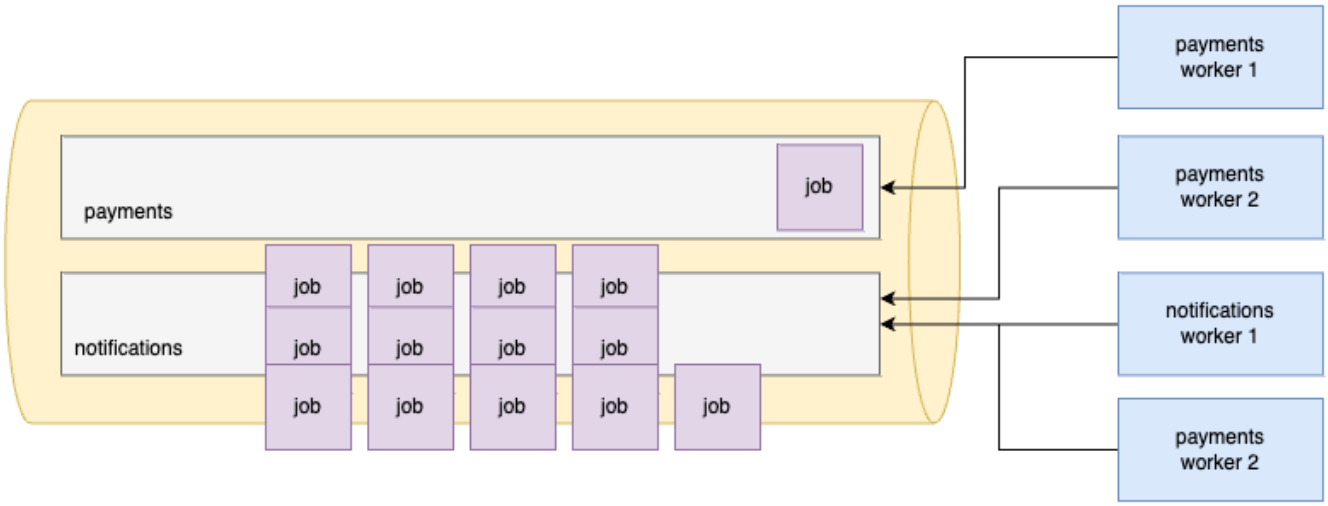
```
php artisan queue:work --queue=payout,notifications
```

The command means that if there are jobs in the payout queue, these workers can **only** process them. However, if the payout queue is empty, they can pick up jobs from the notifications queue as well. And we can do the same for the notifications workers:

```
php artisan queue:work --queue=notifications,payout
```



Now payout workers also pick up jobs from the notifications so we don't waste precious worker processes. But of course, if there are payout jobs they prioritize them over notifications:



That's what we want. And of course, you can have as many queues or workers as you want. In this project, I'll go with only two queues: `payout` and `default`, and two worker processes:

```
php artisan queue:work --queue=payout,default --tries=3 --verbose --
timeout=30 --sleep=3 --max-jobs=1000 --max-time=3600

php artisan queue:work --queue=default,payout --tries=3 --verbose --
timeout=30 --sleep=3 --max-jobs=1000 --max-time=3600
```

In my other book, DevOps with Laravel I write 15 pages exclusively about queues and workers. And another 450 pages about nginx, fpm, docker, docker-compose, kubernetes, swarm, and all that stuff. Check it out here.

So after all that, we successfully sent a payout to a single user. Now it's time to go back to the `SendPayoutsJob` (the one that loops through all the users and dispatches the job we discussed so far) and send them a notification:

```
Bus::chain([
  function () use ($user, $report, $dateService) {
    try {
      ExportTransactionsAction::execute(
        $user,
        $dateService->lastMonth(),
        $report,
      );
    } catch (Exception) {
      $report->status = TransactionReportStatus::Failed;

      $report->save();
    }
  },
  new SendPayoutJob($user, $report, $payoutUuid, $dateService->lastMonth()),
  new SendPayoutNotificationJob($user, $payoutUuid, $report),
])->dispatch();
```

The job itself it pretty simple:

```php
class SendPayoutNotificationJob implements ShouldQueue
{
  use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

  public int $tries = 3;

  public array $backoff = [10, 30];

  public function __construct(
    private readonly User $user,
    private readonly string $payoutUuid,
    private readonly TransactionReport $report,
  ) {}

  public function handle(): void
  {
    $this->user->notify(
      new PayoutSentNotification($this->payoutUuid, $this->report)
    );
  }
}
```

Once again, we use the `$tries` and `$backoff` properties discussed earlier. The Notification is also straightforward:

```php
class PayoutSentNotification extends Notification implements ShouldQueue
{
  use Queueable;

  private Payout $payout;

  public function __construct(
    private readonly string $uuid,
    private readonly TransactionReport $report
  ) {
    $this->payout = Payout::where('uuid', $uuid)->firstOrFail();
  }

  public function via(object $notifiable): array
  {
    return ['mail'];
  }

  public function toMail(object $notifiable): MailMessage
  {
    $amount = MoneyForHuman::from($this->payout->amount);

    $mail = (new MailMessage)
      ->subject('We just sent you money.')
      ->line("We sent you $amount");

    if ($this->report->status === TransactionReportStatus::Done) {
      $mail = $mail->attach(storage_path($this->report->absolutePath()));
    }

    return $mail;
  }
}
```

As I said there can be errors while exporting the report so we only attach it if it's marked as `Done`. In the worst case, users can export it themself using the `transaction-reports` API or you can create a scheduled job that tries to export it again.

The last step is to schedule the payout job:

```
class Kernel extends ConsoleKernel
{
  protected function schedule(Schedule $schedule): void
  {
    if (app()->isProduction()) {
      $schedule->job(SendPayoutsJob::class)->monthly();
    }

    $schedule->job(CleanUpExpiredTransactionReportsJob::class)->daily();
  }
}
```

Dashboard

One of the last features we need is a dashboard where users can see a quick overview of their finances.

The information we want to show them:

- Revenue today
- Revenue this month
- Number of transactions today
- Number of transactions this month
- Average transaction value
- Current balance (money that is going to be paid next time)

This whole dashboard can be a simple `Resource`:

```
class DashboardResource extends JsonResource
{
  public function toArray(Request $request): array
  {
    return [
      'revenue' => [
        'today' => MoneyForHuman::from(
          Transaction::revenue(DateService::today(), $request->user())
        )->value,
        'this_month' => MoneyForHuman::from(
          Transaction::revenue(DateService::thisMonth(), $request->user())
        )->value,
      ],
      'number_of_transactions' => [
        'today' => Transaction::numberOfTransactions(
          DateService::today(), $request->user()
        ),
        'this_month' => Transaction::numberOfTransactions(
          DateService::thisMonth(), $request->user()
        ),
      ],
      'average_transaction_value' => MoneyForHuman::from(
        Transaction::averageTransactionValue($request->user())
      )->value,
      'balance' => MoneyForHuman::from(
        Transaction::balance($request->user())
      )->value,
    ];
  }
}
```

Since the dashboard is quite simple I think having a single resource it's fine. After all, it's just a container that calls other methods and returns a simple array.

I implemented the methods in the `TransactionBuilder`:

```php
class TransactionBuilder extends Builder
{
  public function revenue(DateInterval $interval, User $user): Money
  {
    $sum = $this
      ->where('user_id', $user->id)
      ->whereBetween('created_at', [
        $interval->startDate->date,
        $interval->endDate->date,
      ])
      ->sum('revenue');

    return Money::USD($sum);
  }

  public function numberOfTransactions(
    DateInterval $interval,
    User $user
  ): int {
    return $this
      ->where('user_id', $user->id)
      ->whereBetween('created_at', [
        $interval->startDate->date,
        $interval->endDate->date,
      ])
      ->count();
  }
}
```

As you can see there are a lot of similarities between these two methods and we already used `whereBetween('created_at' [$interval])` multiple times so it's time to write some scopes.

The way you write a scope in a custom Eloquent builder is this:

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

It's like writing any other function but it has to return `self` of course.

The other one:

```
public function whereCreatedBetween(DateInterval $interval): self
{
  return $this->whereBetween('created_at', [
    $interval->startDate->date,
    $interval->endDate->date,
  ]);
}
```

And now the methods become a bit more simple:

```
public function revenue(DateInterval $interval, User $user): Money
{
  $sum = $this
    ->whereUser($user)
    ->whereCreatedBetween($interval)
    ->sum('revenue');

  return Money::USD($sum);
}

public function numberOfTransactions(DateInterval $interval, User $user): int
{
  return $this
    ->whereUser($user)
    ->whereCreatedBetween($interval)
    ->count();
}
```

You can even go one step further and do this:

```
public function revenue(DateInterval $interval, User $user): Money
{
  $sum = $this
     ->transactions()
     ->sum('revenue');

  return Money::USD($sum);
}

public function numberOfTransactions(DateInterval $interval, User $user): int
{
  return $this
     ->transactions()
     ->count();
}

public function transactions(DateInterval $interval, User $user): self
{
  return $this
     ->whereUser($user)
     ->whereCreatedBetween($interval);
}
```

Calculating the average transaction value is also simple:

```
public function averageTransactionValue(User $user): Money
{
  $average = (int) $this
     ->whereUser($user)
     ->average('revenue');

  return Money::USD($average);
}
```

And here's how we can calculate the current balance:

```
public function balance(User $user): Money
{
  $startDate = $user->last_payout?->created_at ?? $user->created_at;

  $balance = $this
     ->whereUser($user)
     ->whereBetween('created_at', [
         $startDate,
         now(),
     ])
     ->sum('revenue');

  return Money::USD($balance);
}
```

If the user has a last payout then that's the start date. If they are a new user we'll use the `created_at` date as a starting point. Then all we need is the sum of the `revenue` column between these dates. Since we don't have a `DateInterval` it's easier to just use `whereBetween` here. However, we can refactor it:

```
$balance = $this
   ->whereUser($user)
   ->whereCreatedBetween(DateInterval::make(
     StartDate::fromCarbon($startDate),
     EndDate::fromCarbon(now()),
   ))
   ->sum('revenue');
```

It's not that bad, but we can go a step further and hide this `DateInterval` creation in the `DateInterval` class itself:

```php
namespace App\ValueObjects;

class DateInterval
{
  public function __construct(
    public StartDate $startDate,
    public EndDate $endDate,
  ) {}

  public static function make(StartDate $startDate, EndDate $endDate): self
  {
    return new self($startDate, $endDate);
  }

  public static function fromCarbons(Carbon $startDate, Carbon $endDate): self
  {
    return new self(
      StartDate::fromCarbon($startDate),
      EndDate::fromCarbon($endDate),
    );
  }
}
```

And now the query looks much simpler:

```php
$balance = $this
  ->whereUser($user)
  ->whereCreatedBetween(DateInterval::fromCarbons($startDate, now()))
  ->sum('revenue');
```

And I think this is the perfect time to ask ourselves the question:

What did we accomplish by refactoring this

```
$balance = $this
   ->whereUser($user)
   ->whereBetween('created_at', [
       $startDate,
       now(),
   ])
   ->sum('revenue');
```

Into this?

```
$balance = $this
   ->whereUser($user)
   ->whereCreatedBetween(DateInterval::fromCarbons($startDate, now()))
   ->sum('revenue');
```

In my honest opinion, the answer is: nothing. Or, nothing meaningful. Users don't care if it's `whereBetween` or `whereCreatedBetween` with fine value objects. Owners don't care about it. If it's your side project, your Stripe account doesn't care about it. Of course, it's an extreme example. What I'm trying to say is that there's a line between writing good code, and being a snob and wasting your time on micro stuff that doesn't matter a lot in the long term. For example, this is already quality code, in my opinion:

```
$balance = $this
   ->whereUser($user)
   ->whereBetween('created_at', [
       $startDate,
       now(),
   ])
   ->sum('revenue');
```

The point is that we like to overdo stuff. Especially when we first learn about something new. Just try to remember that, and be reasonable when it comes to refactoring and code quality. And I also think if you're working on your own side project, it's extremely important not to overthink stuff.

Backups

There's one thing every project needs. It's regular backups of your database and files. Fortunately, it's pretty easy Spatie's amazing package spatie/laravel-backup.

After you've installed the package, you can configure it in the `config/backup.php` config file:

```
'source' => [
  'files' => [
    'include' => [
        base_path(),
    ],
    'exclude' => [
      base_path('vendor'),
      base_path('node_modules'),
    ],
  ],
],
```

This is the default configuration that works well for lots of projects. It includes every file of your application except the `vendor` and `node_modules` folders.

After that, you need to run:

```
php artisan backup:run
```

And you'll have your backup in the `storage` folder by default. Of course, in production, it's not a good idea to store backups on the same server, so it's better to store them on S3 for example.

To do that you need to set up your destination:

```
'destination' => [
  'filename_prefix' => '',

  'disks' => [
    'local',
    's3',
  ],
],
```

Now laravel-backup will store the backup on your local storage and S3 as well. You also need to configure S3 credentials:

```
AWS_ACCESS_KEY_ID= ...
AWS_SECRET_ACCESS_KEY= ...
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=backups
AWS_USE_PATH_STYLE_ENDPOINT=false
```

The last thing we need to do is to schedule the `backup:run` command:

```
protected function schedule(Schedule $schedule): void
{
  if (app()->isProduction()) {
    $schedule->job(SendPayoutsJob::class)->monthly();

    $schedule->command('backup:run')->daily();
  }

  $schedule->job(CleanUpExpiredTransactionReportsJob::class)->daily();
}
```

Activity logs

Another crucial thing to have in all applications is an activity or audit log. This means you log everything in your application. For example, if a user updates a product's price you create records such as this:

id	subject_type	subject_id	event	old_data	new_data
1	App\Models\Product	1	updated	{"id": 1, "price": 1900}	{"id": 1, "price": 2900}

So you log every change in the application including these events:

- Create
- Update
- Delete

This way you have the complete history of your application in one table that can be used for mainly debugging.

Fortunately, Spatie (as always) has an excellent package called laravel-activitylog. After you've installed the package, you can configure it in the `config/activitylog.php` config file:

```
return [
  'delete_records_older_than_days' => 365,

  'default_log_name' => 'default',

  'activity_model' => \Spatie\Activitylog\Models\Activity::class,

  'table_name' => 'activity_log',

  'database_connection' => env('ACTIVITY_LOGGER_DB_CONNECTION'),
];
```

These are some of the most important configs. By default, it uses the `activity_log` table and the same connection as your application. If you want you can have a separate activity log database.

To enable it on a model, all we need to do is this:

```php
namespace App\Models;

class Product extends Model
{
  use LogsActivity;

  public function getActivitylogOptions(): LogOptions
  {
    return LogOptions::defaults()
      ->logFillable();
  }
}
```

We need to use the `LogsActivity` trait. In the `getActivitylogOptions` you can customize how the package should log your activity. In this case, I'm using all the default options, and it logs every change that affects some of the columns in the `$fillable` array. You can read more about log options here.

As a result, we have a table such as this:

id	log_name	description	subject_type	event	subject_id	causer_type	causer_id	properties	batch_uuid	created_at	updated_at
397	default	created	App\Models\Transaction	created	350	NULL	NULL	{"attributes": {"revenue": {"amount": "6886",…	NULL	2024-0…	2024-01…
398	default	created	App\Models\Webhook	created	13	NULL	NULL	{"attributes": {"url": "https://abernathy.org/…	NULL	2024-0…	2024-01…
399	default	created	App\Models\Webhook	created	14	NULL	NULL	{"attributes": {"url": "http://dibbert.com/eni…	NULL	2024-0…	2024-01…
400	default	created	App\Models\Product	created	36	NULL	NULL	{"attributes": {"url": "http://www.miller.com/…	NULL	2024-0…	2024-01…
401	default	created	App\Models\Product	created	37	NULL	NULL	{"attributes": {"url": "http://zemlak.com/est-…	NULL	2024-0…	2024-01…
402	default	created	App\Models\Product	created	38	NULL	NULL	{"attributes": {"url": "http://grady.info/et-s…	NULL	2024-0…	2024-01…
403	default	created	App\Models\Product	created	39	NULL	NULL	{"attributes": {"url": "http://www.quigley.com…	NULL	2024-0…	2024-01…
404	default	created	App\Models\Product	created	40	NULL	NULL	{"attributes": {"url": "http://www.funk.com/ne…	NULL	2024-0…	2024-01…
405	default	created	App\Models\Transaction	created	351	NULL	NULL	{"attributes": {"revenue": {"amount": "1427",…	NULL	2024-0…	2024-01…
406	default	created	App\Models\Transaction	created	352	NULL	NULL	{"attributes": {"revenue": {"amount": "6728",…	NULL	2024-0…	2024-01…
407	default	created	App\Models\Transaction	created	353	NULL	NULL	{"attributes": {"revenue": {"amount": "7295",…	NULL	2024-0…	2024-01…
408	default	created	App\Models\Transaction	created	354	NULL	NULL	{"attributes": {"revenue": {"amount": "1458",…	NULL	2024-0…	2024-01…
409	default	created	App\Models\Transaction	created	355	NULL	NULL	{"attributes": {"revenue": {"amount": "8451",…	NULL	2024-0…	2024-01…
410	default	created	App\Models\Transaction	created	356	NULL	NULL	{"attributes": {"revenue": {"amount": "3923",…	NULL	2024-0…	2024-01…

There's only one thing we need to do. Cleaning old log entries. You can configure how old a record should be to be deleted in the `delete_records_older_than_days` config. By default, it's `365`. To delete old logs we need to schedule the `activitylog:clean` command:

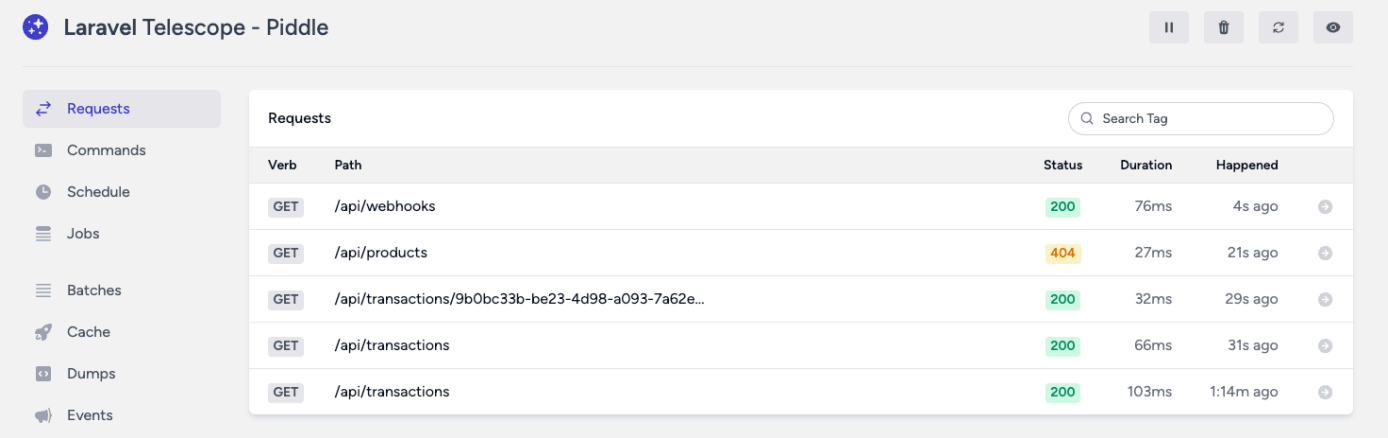
```php
namespace App\Console;

class Kernel extends ConsoleKernel
{
  protected function schedule(Schedule $schedule): void
  {
    if (app()->isProduction()) {
      $schedule->job(SendPayoutsJob::class)->monthly();

      $schedule->command('backup:run')->daily();
    }

    $schedule->job(CleanUpExpiredTransactionReportsJob::class)->daily();

    $schedule->command('activitylog:clean')->daily();
  }
}
```

Basic performance optimizations

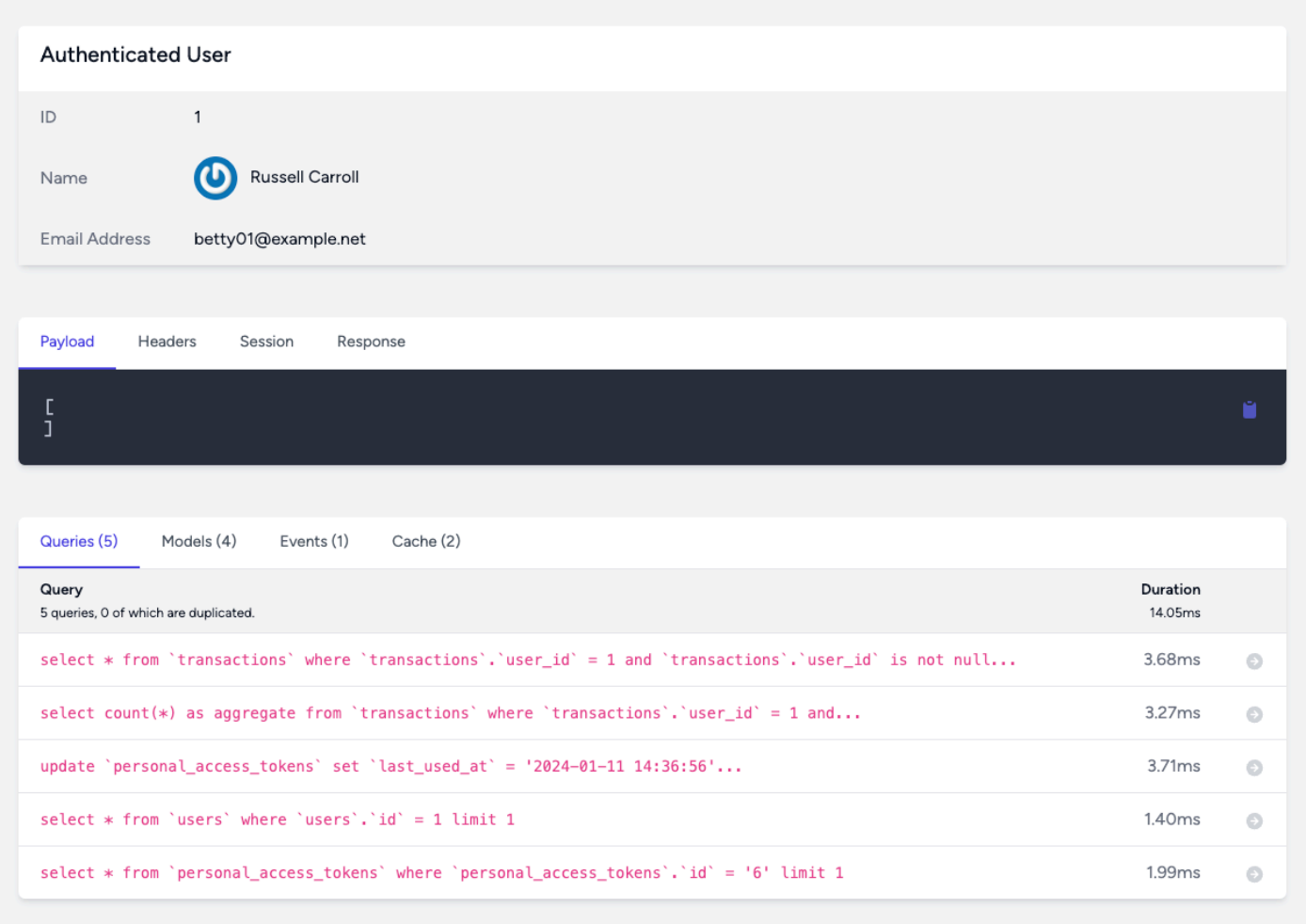
If you're not really familiar with MySQL indices please first read this article.

In my opinion, in every project, you should spend some time on optimizing performance and making your application a bit more faster/responsive. One of the easiest tools you can use to monitor your app is Laravel Telescope.

After you've installed the package you can access the dashboard at `localhost:8000/telescope`. If you send some requests you'll see something like that:

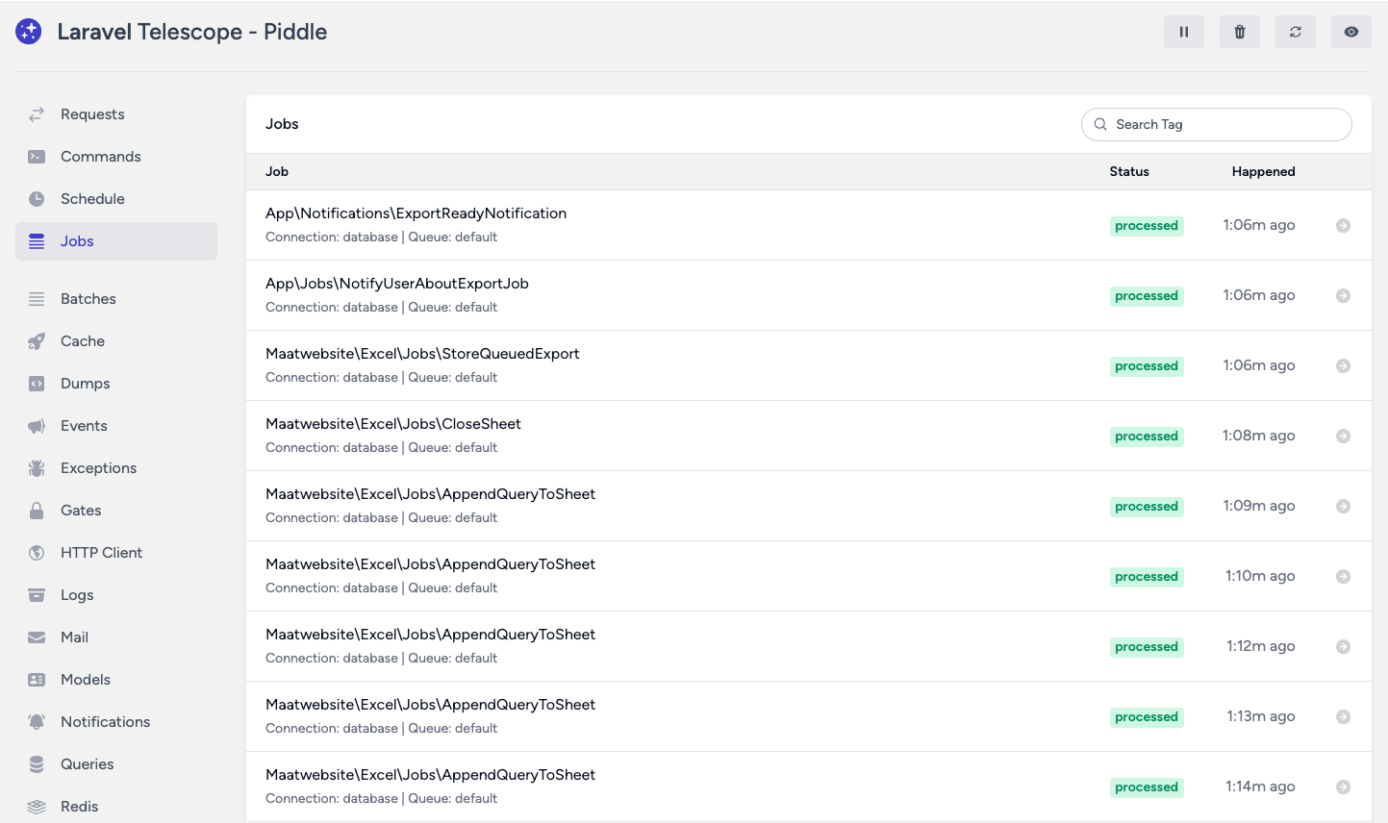


It gives you a great overview of your requests and their duration. What's even better, if you click on a specific request you can see all the database queries that were executed:

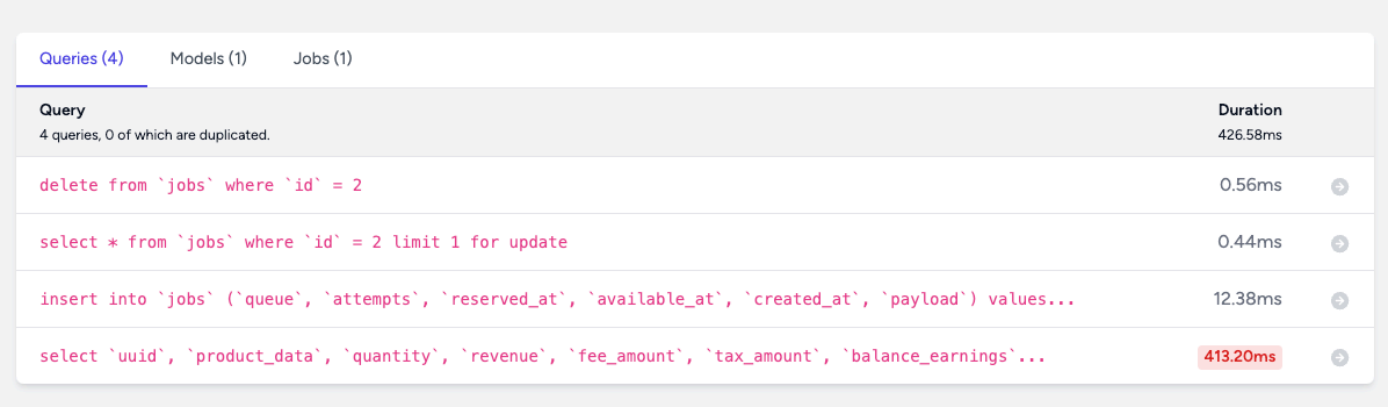


Exporting transaction reports

Let's export a transaction report and see how many queries the app runs. In this case, we don't want to check requests but jobs instead:



There are lots of `AppendQueryToSheet` jobs. 40 to be exact. That's because we process 250 rows in one chunk and the current user 10 000 transactions. If we check the queries they look like this:



There are only 4 queries and 3 of them were made by Laravel itself. There's only one query we're running directly:

```
select
  `uuid`,
  `product_data`,
  `quantity`,
  `revenue`,
  `fee_amount`,
  `tax_amount`,
  `balance_earnings`,
  `customer_email`,
  `created_at`
from
  `transactions`
where
  `user_id` = 1
  and `created_at` between '2023-10-01 00:00:00'
  and '2024-01-11 23:59:59'
order by
  `created_at` asc
limit
  250 offset 0
```

And as you can see it took **413ms**. It's a terrible **huge** number for a database query. This query runs 40 times so it takes 16 seconds to get 10 000 rows from a table with 200 000 rows. 413ms is huge, 16 seconds is huge, 10,000 is small, 200,000 is small. Something is clearly wrong so let's improve it.

When I wrote the migration for the `transactions` table I added two standard indices:

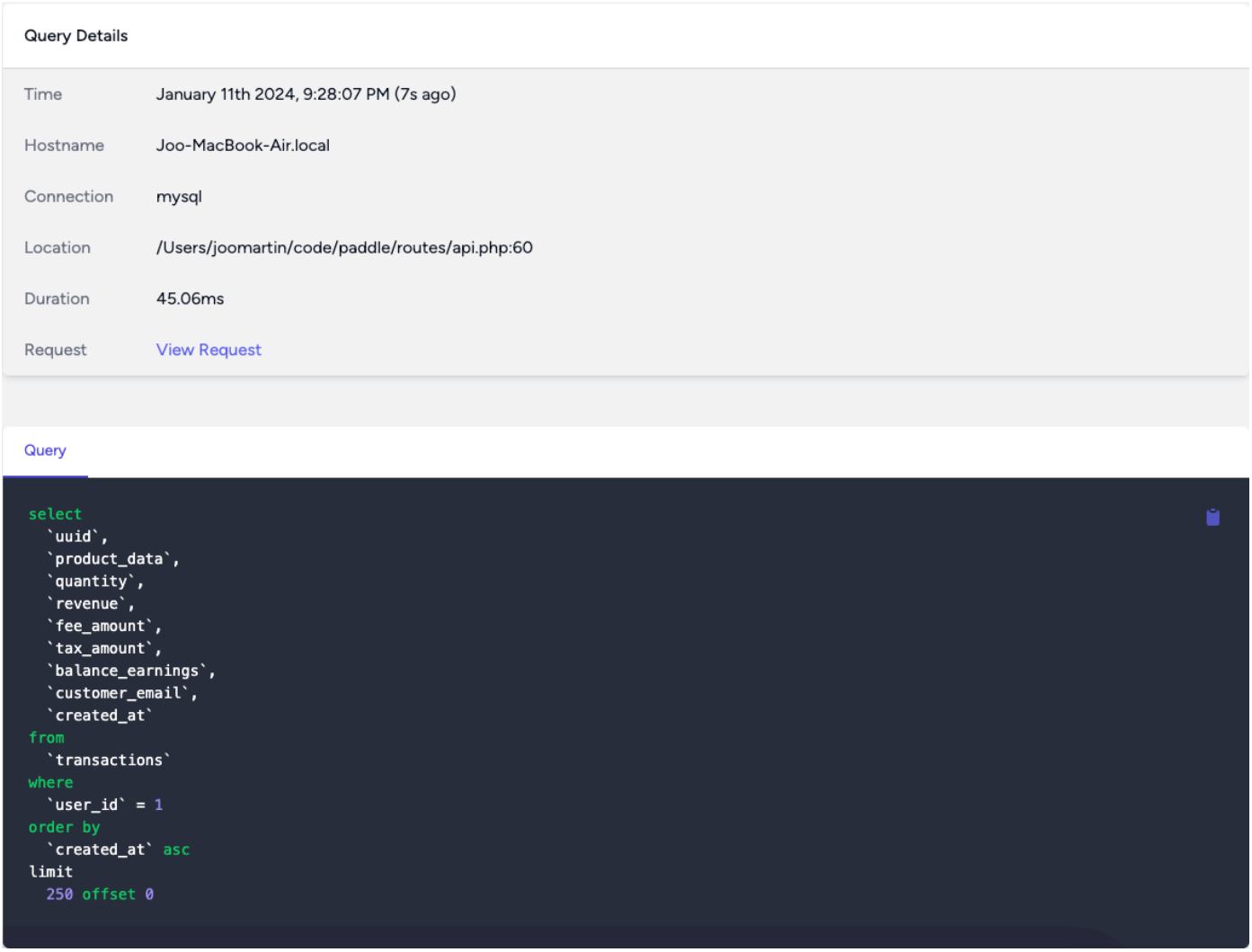
```
Schema::create('transactions', function (Blueprint $table) {
  $table->id();
  $table->unsignedBigInteger('user_id');
  $table->timestamps();
  // ...

  $table->index('user_id');

  $table->index('created_at');
});
```

So there are two separate indices for the two columns. I think that's a common default choice for lots of developers.

Let's modify the query for a minute and filter it only by the user ID:

```
select
  `...`
from
  `transactions`
where
  `user_id` = 1
order by
  `created_at` asc
limit
  250 offset 0
```

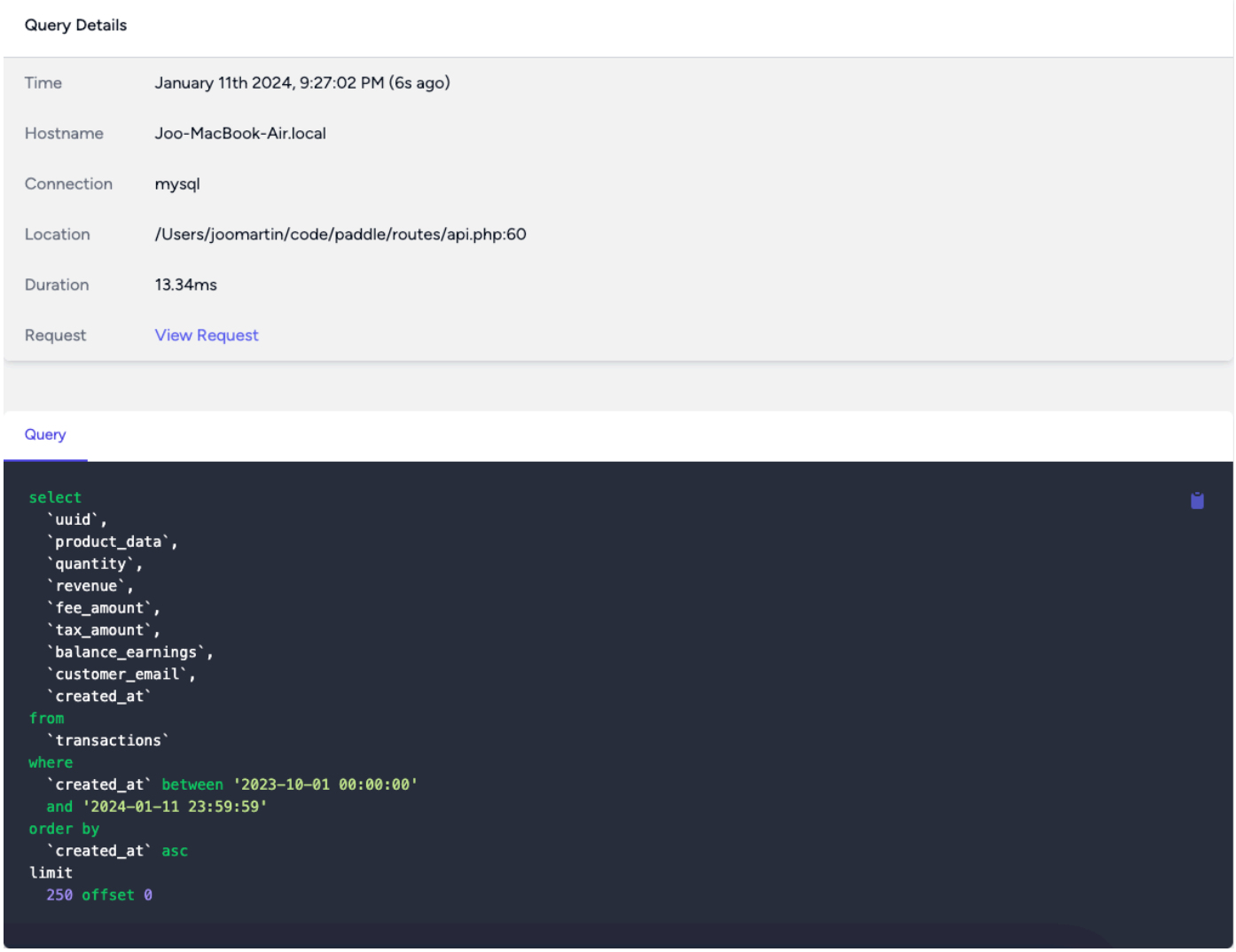
Now the query takes only **45ms**:



Interesting. What happens if we leave only the `create_at` where expression?

```
select
  `...`
from
  `transactions`
where
 `created_at` between '2023-10-01 00:00:00'
  and '2024-01-11 23:59:59'
order by
  `created_at` asc
limit
  250 offset 0
```

Now the query takes 13ms:



Let's dig deeper by running `explain` on the original query:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	transactions	NULL	range	user_id,created_at	created_at	5	NULL	93909	9.88	Using index condition; Using where

- Type `range` means that MySQL was able to limit the number of rows that it has to go through. So it doesn't need to scan the full table. It's a good thing.
- `possible_keys` means MySQL considered using either the `user_id` or the `created_at` keys.
- `key` means that it decided to use the `created_at` key
- `rows` means it went through `93 909`.

Scanning only 94k rows is a good thing if we consider that there are 200k rows in the table but it's a bad thing if we know that the given user has only 10k transactions. All of this is because MySQL used the `created_at` index. And it used this index because we have the `order by created_at asc`. In the `created_at` index (which is a B-TREE but you can imagine it as another table) there are 200k rows sorted by the `created_at` date, something like that:

created_at	row_id
2023-12-01	2
2023-12-02	1
2023-12-03	3

`row_id` refers to a row in the `transactions` table. It's an internal ID, not our primary key. Using this index all that MySQL can do is to filter rows between 2023-10-01 and 2024-01-11. Which results in roughly 94k rows. And then it goes through them reads the the rows from your disk and filters the final 10k rows based on the `user_id`.

If I delete the `order by` and run `explain` again this is the result:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	transactions	NULL	ref	user_id,created_at	user_id	8	con…	18560	50.00	Using where

Now it uses the `user_id` index and scans only 18k rows.

The big conclusion so far is this: **database indices do not exist in a vacuum.** They are specific to queries!

There's no thing such as "Hey I'll just add an index to the created_at column because you know... that's my default behavior as a developer. An index is always a good thing, isn't it?"

In this specific case, if I delete the created_at index I'll get better results!

You want to add indices based on the nature of your queries and features. For example, in this demo application most of the where expressions use both the `user_id` and the `created_at` columns as well.

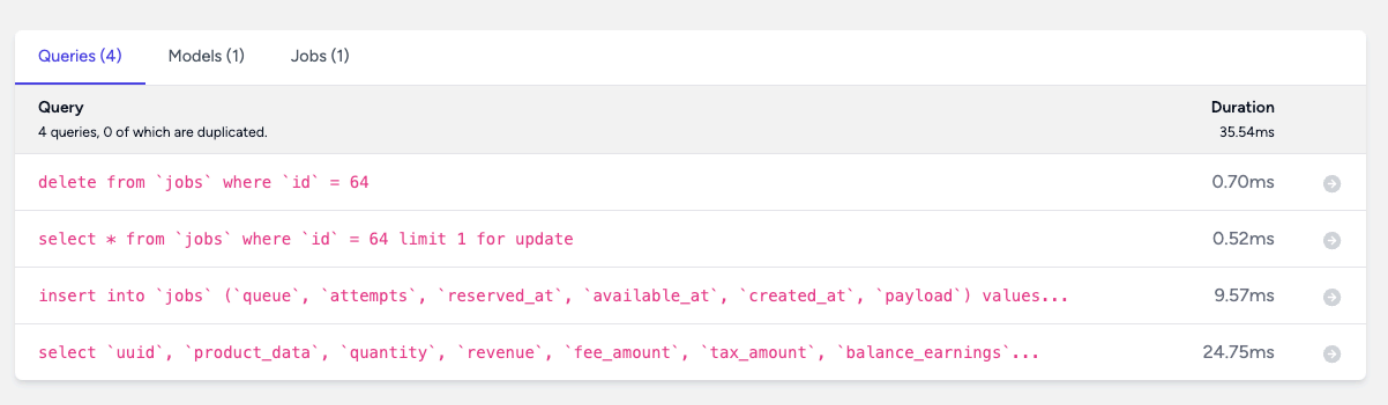
Let's try a composite index!

```
CREATE INDEX user_id_created_at_idx ON transactions (user_id, created_at);
```

Let's run explain:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	transactions	NULL	range	user_id_created_a…	user_id_created_at_idx	13	NULL	18962	100.00	Using index condition

`range` type using the composite index scanning 18k rows. Check out Telescope:



It took only 24ms. There are some instances where it took 6ms and others where it took 40ms. Of course, it varies but the difference is huge.

Before the composite index, the numbers were:

- **413ms** per job
- 40 jobs so **16s**

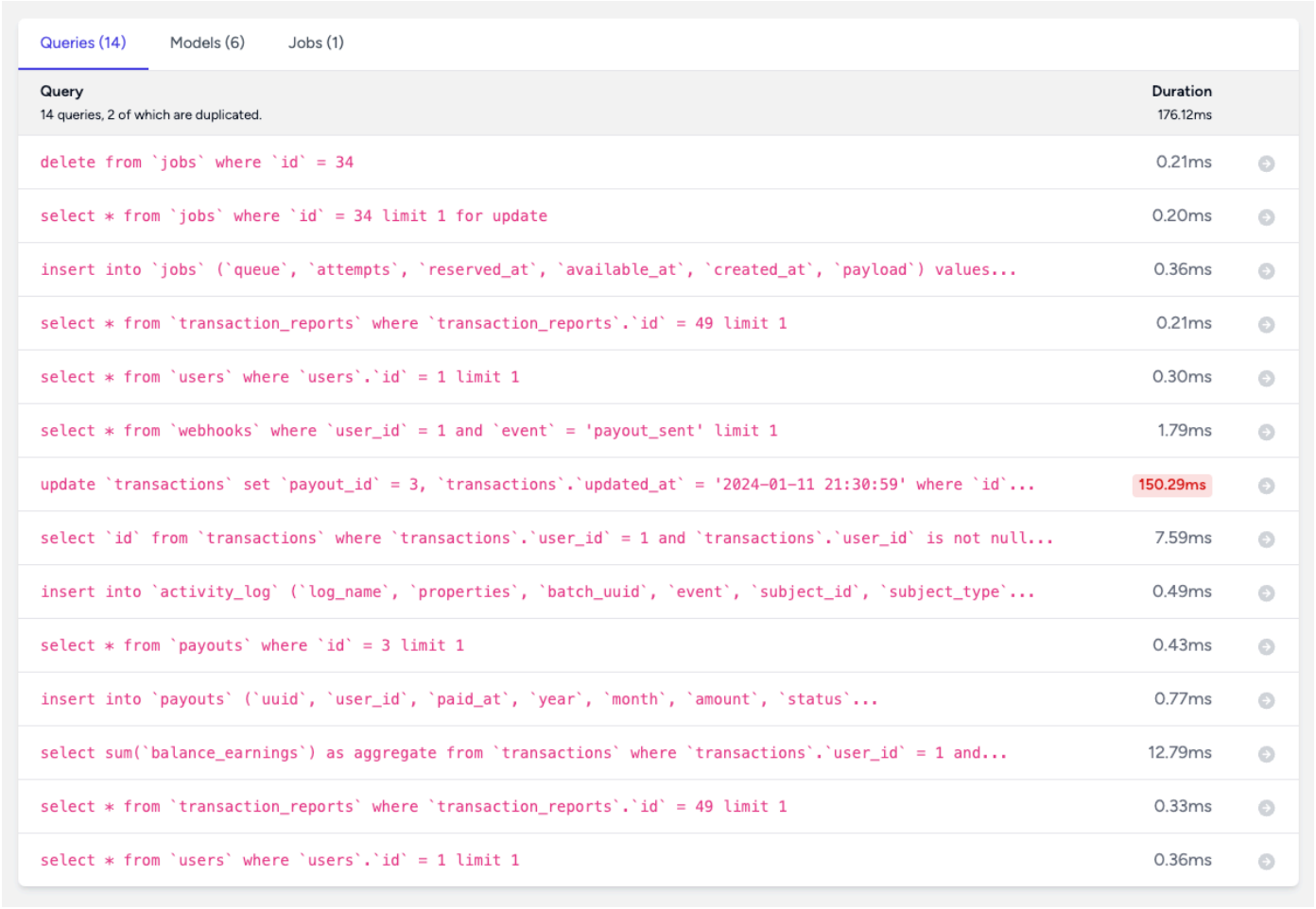
After the composite index:

- 24-50ms per job
- 40 jobs so 1-2s

This one index resulted in an ~8-10x performance improvement. And it took only 40-60 minutes.

Sending payout

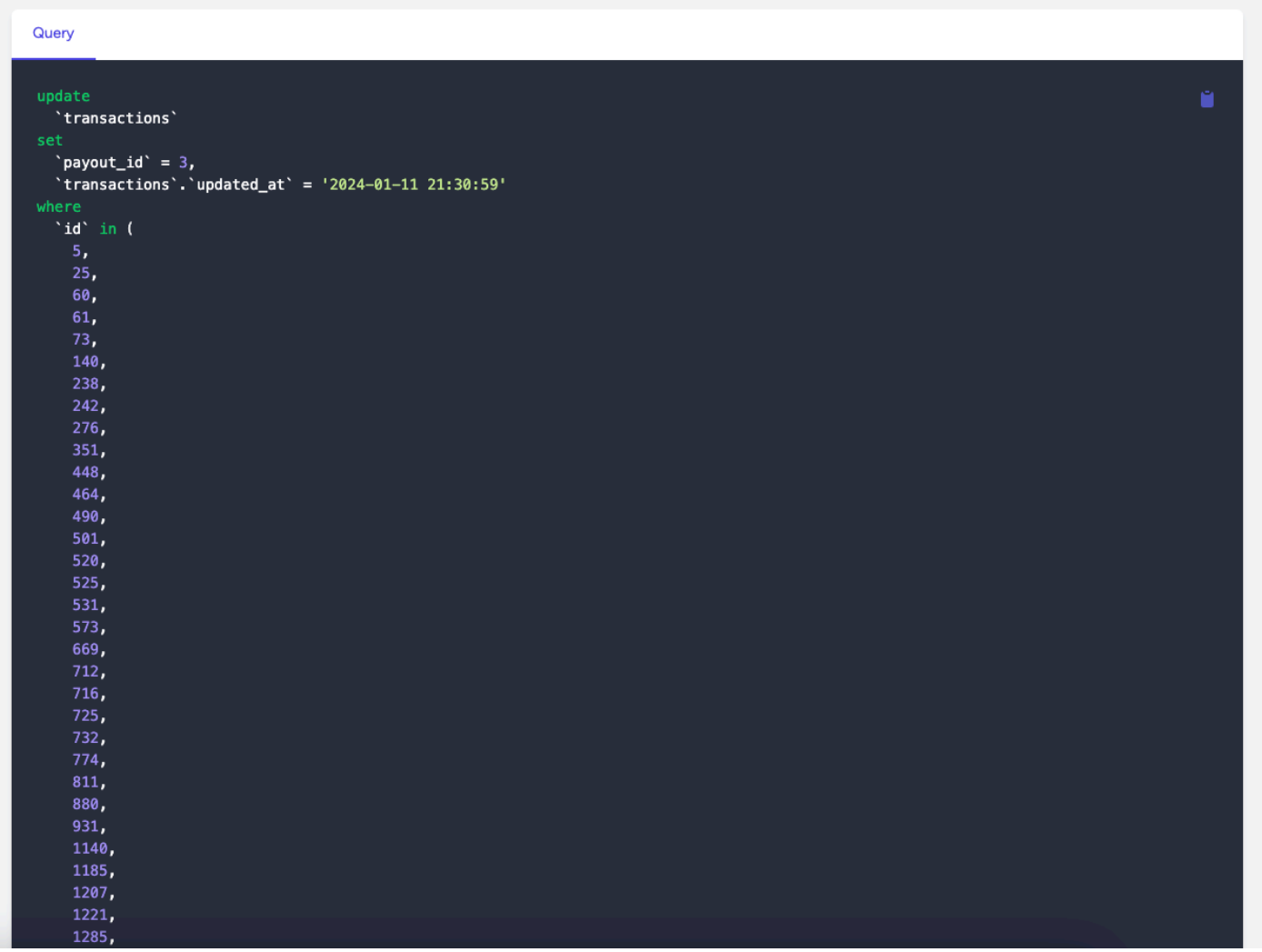
Probably the most important feature of this application is sending payouts. So let's do that and see what we can improve. These are the queries in a `SendPayoutJob`:



Queries (14) | Models (6) | Jobs (1)

Query 14 queries, 2 of which are duplicated.	Duration 176.12ms
`delete from `jobs` where `id` = 34`	0.21ms
`select * from `jobs` where `id` = 34 limit 1 for update`	0.20ms
`insert into `jobs` (`queue`, `attempts`, `reserved_at`, `available_at`, `created_at`, `payload`) values...`	0.36ms
`select * from `transaction_reports` where `transaction_reports`.`id` = 49 limit 1`	0.21ms
`select * from `users` where `users`.`id` = 1 limit 1`	0.30ms
`select * from `webhooks` where `user_id` = 1 and `event` = 'payout_sent' limit 1`	1.79ms
`update `transactions` set `payout_id` = 3, `transactions`.`updated_at` = '2024-01-11 21:30:59' where `id`...`	150.29ms
`select `id` from `transactions` where `transactions`.`user_id` = 1 and `transactions`.`user_id` is not null...`	7.59ms
`insert into `activity_log` (`log_name`, `properties`, `batch_uuid`, `event`, `subject_id`, `subject_type`...`	0.49ms
`select * from `payouts` where `id` = 3 limit 1`	0.43ms
`insert into `payouts` (`uuid`, `user_id`, `paid_at`, `year`, `month`, `amount`, `status`...`	0.77ms
`select sum(`balance_earnings`) as aggregate from `transactions` where `transactions`.`user_id` = 1 and...`	12.79ms
`select * from `transaction_reports` where `transaction_reports`.`id` = 49 limit 1`	0.33ms
`select * from `users` where `users`.`id` = 1 limit 1`	0.36ms

All in all, we execute 14 queries per user. 3 of them were executed by Laravel. It's not that bad in my opinion. But there's one query that took 150ms. And it's an update. Interesting. Let's see what's inside:



This is the query that updates the `payout_id` column for a given user's transaction. In this test, my user had ~6,000 transactions last month so there are 6,000 IDs in the where expression.

In a real application, now you need to decide what you want to do:

- If an average user has 6k transactions a month then it's a common problem. So you can probably fix it.
- If only 1% of users have this volume it's probably not a big problem and you can ignore it.

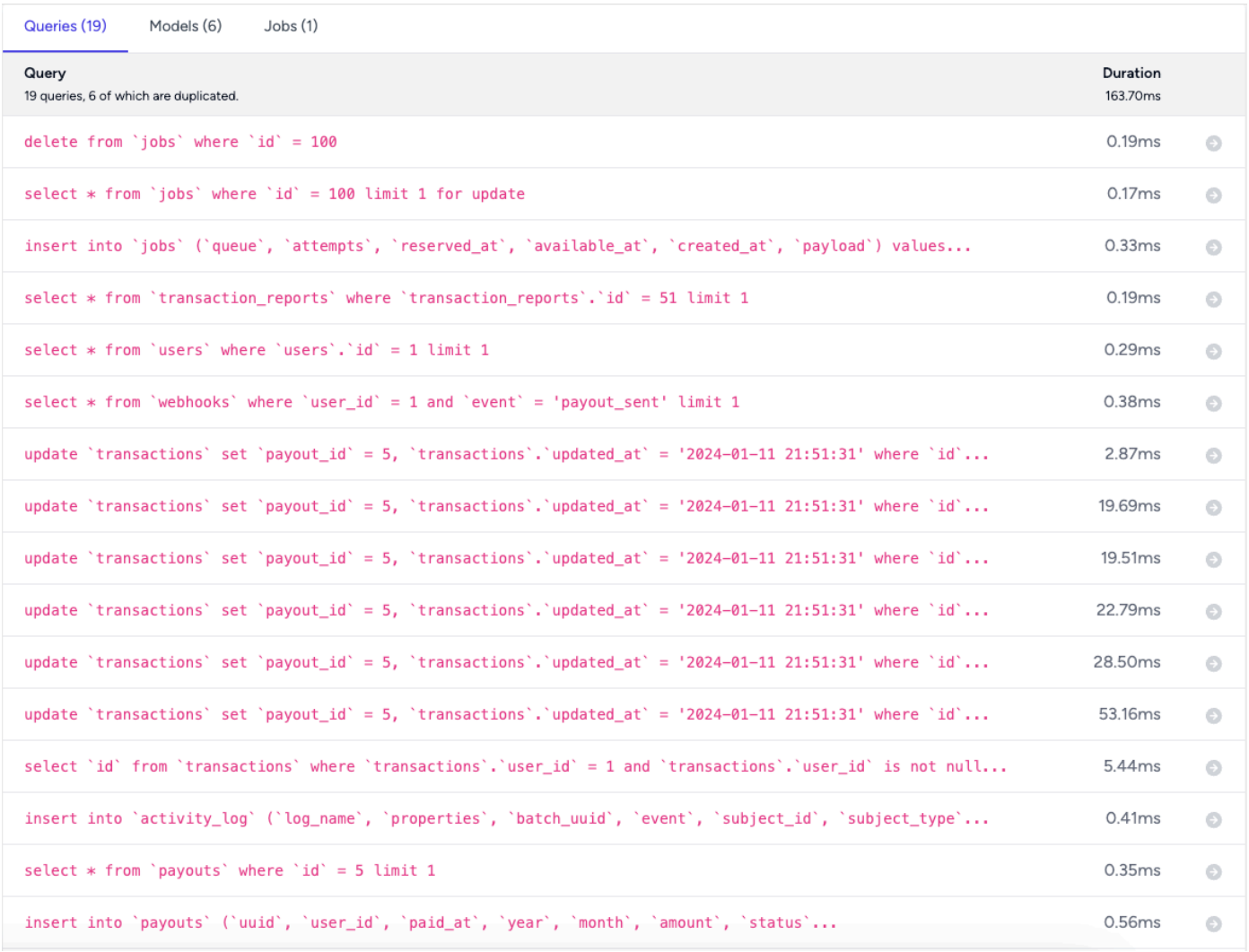
Let's say in our demo app this is an average user, so we want to improve the situation. This is the code that performs the update:

```
$transactionIds = $query->select('id')->pluck('id');

Transaction::query()
    ->whereIn('id', $transactionIds)
    ->update(['payout_id' => $payout->id]);
```

There's a pretty simple solution: we can chunk the collection

```php
$transactionIds = $query->select('id')->pluck('id');

$chunks = $transactionIds->chunk(1000);

foreach ($chunks as $chunk) {
  Transaction::query()
    ->whereIn('id', $chunk)
    ->update(['payout_id' => $payout->id]);
}
```

Now it's only updating 1,000 transactions at a time. So it results in smaller and faster queries:



Queries (19) | Models (6) | Jobs (1)

Query 19 queries, 6 of which are duplicated.	Duration 163.70ms
delete from `jobs` where `id` = 100	0.19ms
select * from `jobs` where `id` = 100 limit 1 for update	0.17ms
insert into `jobs` (`queue`, `attempts`, `reserved_at`, `available_at`, `created_at`, `payload`) values...	0.33ms
select * from `transaction_reports` where `transaction_reports`.`id` = 51 limit 1	0.19ms
select * from `users` where `users`.`id` = 1 limit 1	0.29ms
select * from `webhooks` where `user_id` = 1 and `event` = 'payout_sent' limit 1	0.38ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	2.87ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	19.69ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	19.51ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	22.79ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	28.50ms
update `transactions` set `payout_id` = 5, `transactions`.`updated_at` = '2024-01-11 21:51:31' where `id`...	53.16ms
select `id` from `transactions` where `transactions`.`user_id` = 1 and `transactions`.`user_id` is not null...	5.44ms
insert into `activity_log` (`log_name`, `properties`, `batch_uuid`, `event`, `subject_id`, `subject_type`...	0.41ms
select * from `payouts` where `id` = 5 limit 1	0.35ms
insert into `payouts` (`uuid`, `user_id`, `paid_at`, `year`, `month`, `amount`, `status`...	0.56ms

Of course, we have more queries but they take less time and probably less memory and less overhead for MySQL. In general, it's always a good idea to split big stuff into smaller pieces. It can be a huge class, a long method, a big monolith, or a pretty long update query.

All right, these were the two most obvious performance problems in the application and we successfully solved them. If you want to learn more about performance check out my upcoming book Performance with Laravel.