SECOND EDITION

LARAVEL QUEUES IN ACTION

MOHAMED SAID

Contents

Preface

Hey, welcome to *Laravel Queues in Action*! I'm glad you're reading this book and excited to join you in exploring the many ways we can use the queue system to enhance our applications.

Over the years, I worked on projects that heavily relied on asynchronous task execution. Also, during my time at Laravel, I worked with hundreds of developers to solve problems that involved the queue system. Along the way, I contributed to enhancing the system by adding new features, fixing bugs, and improving performance.

While the Laravel community was growing, we collectively discovered best practices for working with the queue system. This knowledge was shared in separate blog posts here and there. But there still wasn't a comprehensive guide on utilizing its power and dealing with common problems in the community.

It was with this in mind that I decided to write this book in 2020. Fast forward to 2022, several Laravel releases came out, and more usage patterns were discovered. These new releases introduced several enhancements to the system that dealt with many performance and developer experience issues.

In this second edition, I think that the book is now about giving you the information you need to make educated decisions about how to best use the queue system in your projects. I have done my best to make the book insightful to first-time queue system explorers and long-time professionals.

I hope that *Laravel Queues in Action* proves helpful!

CHAPTER 1

Introduction to Queues

A web application is a series of procedures that pass data around. It takes data in, performs several operations, and passes it back out. That makes a web application a form of data *pipeline*. The term pipeline is borrowed from industry in which pipes transport substances from one place to another. In the context of web applications, each pipe—sometimes referred to as *stage*—in a pipeline performs an operation that validates, transforms, stores, or transports the data. Other stages may not touch the data, but they take action based on it.

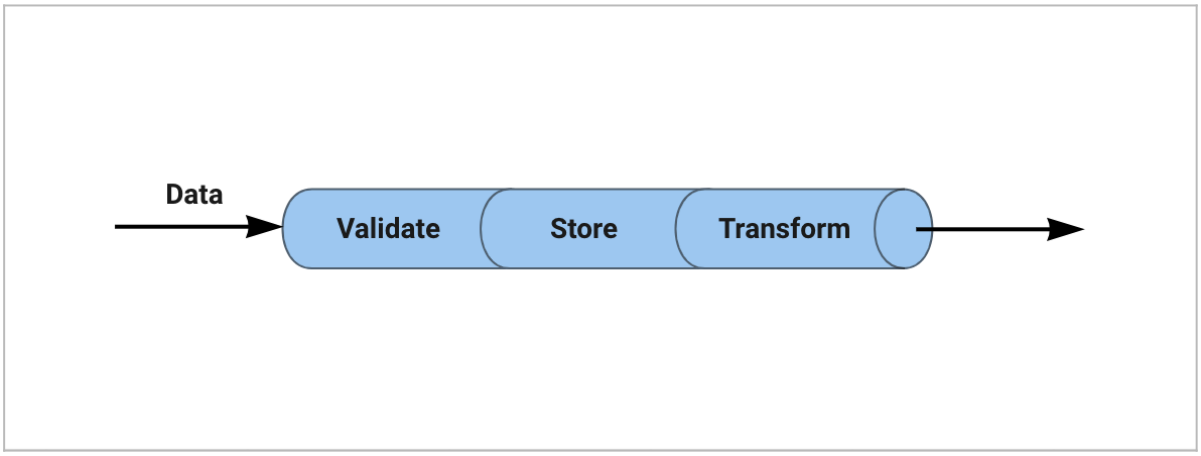

Figure 1-1. A data pipeline

Like an industrial pipeline, the data needs to pass through one pipe to move to the next, and each pipe has a resistance coefficient. The higher the coefficient, the slower the data will move in a pipe. Therefore, adding more pipes with high resistance will result in an overall delay in passing the data out of the final pipe.

Take a look at this example controller action:

```
public function store(Request $request)
{
    $validated = $this->validate($request, [
        'email' => 'email'
    ]);
```

```
    $user = User::create($validated);

    Mail::to($user)->send(new Welcome());

    return UserResource::make($user);
}
```

In this data pipeline, the data is passed from the request to the validation stage, the storage stage, the mail sending stage, and the transformation stage before it's passed out to the client making the request. Each stage's resistance coefficient dictates how fast the data will pass through. The slower the operation, the longer it takes for the response to be returned to the client.

But unlike industrial pipelines, a single web application instance can only handle one request at any time. If more requests come, they will have to wait until the current request is passed out of the final pipe. That means slow operations not only delay the response of one request but also delay handling other requests. For example, if the machine that hosts your web application can run fifty instances of the web application and all fifty slots are occupied, new requests will have to wait.

For a business to be able to handle more traffic, it has three options:

1. Buy more machines (called scaling out).
2. Buy more computing resources for the existing machines (called scaling up).
3. Task software engineers to decrease the resistance coefficient of all the stages of handling a request.

Sometimes throwing money at the problem–buying more computing resources–is more efficient. Other times putting more software engineering efforts is more effecient.

Decreasing the resistance can be achieved in various ways:

1. Code performance optimization (Figure 1-2).
2. Concurrent processing; running several stages in parallel (Figure 1-3).
3. Asynchronous execution; removing some stages from the main pipeline and putting it into another pipeline (Figure 1-4).

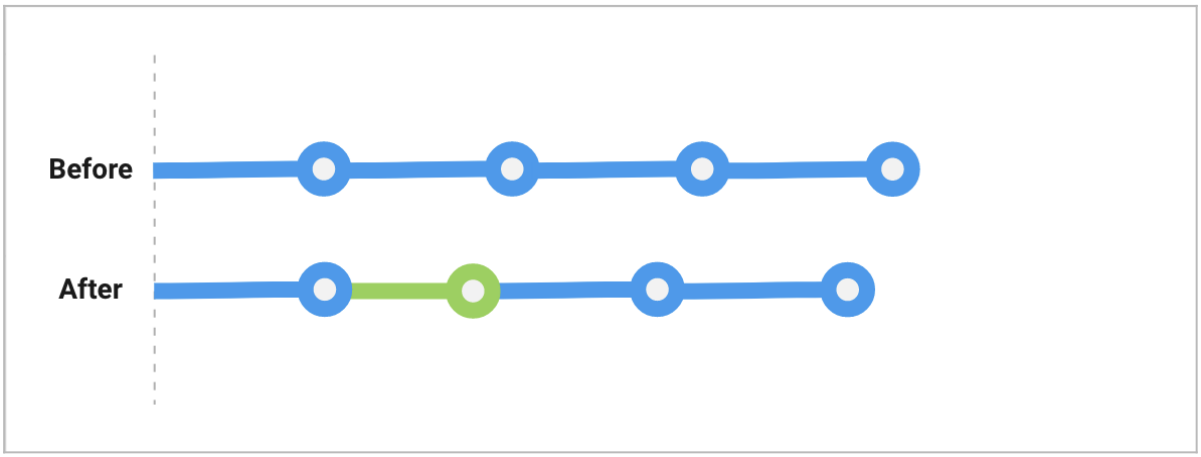

Figure 1-2. The 2nd stage was optimized for performance

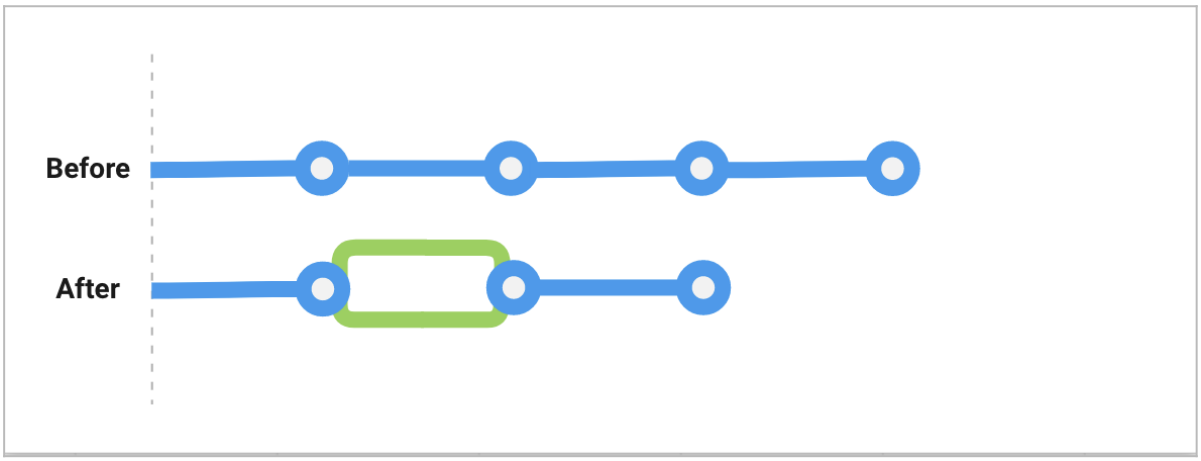

Figure 1-3. The 2nd & 3rd stages were executed concurrently

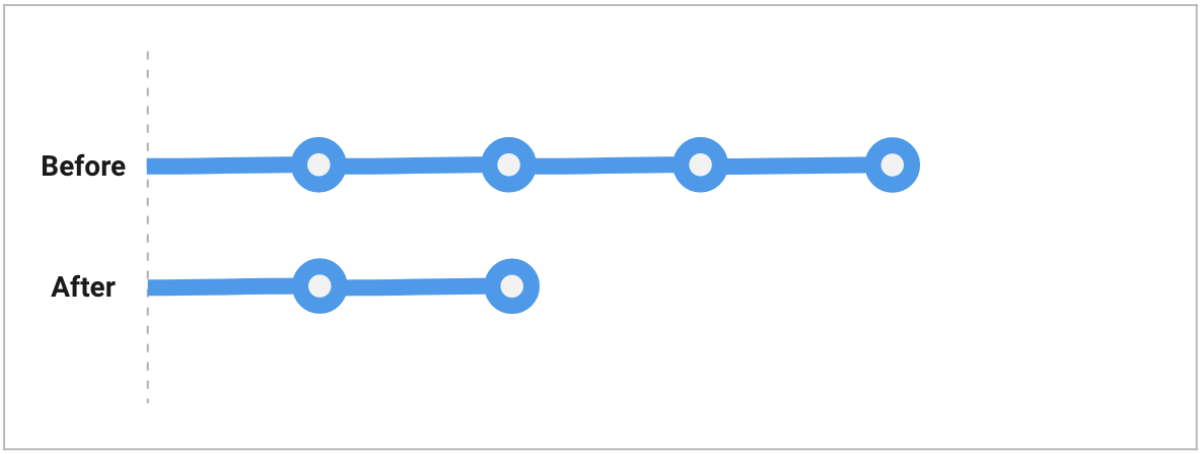

Figure 1-4. The 2nd & 3rd stages were moved to a different pipeline

Laravel is a web framework that ships with many tools that help us monitor and enhance the performance of our code. It also provides us with tools we can use for concurrent processing and asynchronous execution.

Concurrent Execution

For stages that need to complete before sending the response back to the user, we may install Laravel Octane and use the Swoole client. Doing so will allow us to split an operation into several smaller ones, execute them concurrently, and wait for the result before sending it to the user.

This pattern is called *fan-out, fan-in*. Fanning out is the process of distributing work on multiple processors, and fanning in is combining the results of those processes into one set and passing it to the next stage of the pipeline.

Concurrent execution in PHP is not supported out of the box. However, the Swoole (and its fork *Open Swoole*) extension allows us to use multiprocessing to send commands from our main process (pipeline) to one or more worker processes and wait for the results. Laravel Octane utilizes this and abstracts it under the `concurrently()` method:

```
public function store(Request $request)
{
    $request->user()->can('load_entities');

    [$users, $servers] = Octane::concurrently([
        fn () => User::all(),
        fn () => Supplier::all(),
    ]);

    return [$users, $servers];
}
```

In this example, the authorization stage will ensure the current logged-in user is allowed to `load_entities`, then using `Octane::concurrently()`, we can fan out the tasks of loading users and suppliers to two worker processes to work on them in parallel. Once the results from the two processes come back, they will be moved to the transformation stage before responding to the web client.

Octane's concurrency model helps us run several stages of our program simultaneously and shorten the time needed to handle the request. But what if we can move some of the stages completely from the pipeline to a different pipeline? That way, we can remove all the work that doesn't need to happen synchronously–while handling the request–and execute it asynchronously in the background.

Take another look at this example:

```
$user = User::create($validated);

Mail::to($user)->send(new Welcome());

return UserResource::make($user);
```

We don't have to pass the request through the mail-sending stage before it reaches the transformation stage. Instead, we can pass the request directly from the storage stage to the transformation stage and move the mail sending stage to an asynchronous pipeline to be done later.

That's what the queue system that ships with Laravel is for.

The Queue System

A user interface is how a human interacts with a program, and a programmable interface (API) is how a program interacts with another program. A queue system offers a programmable interface that programs use to communicate asynchronously, not in real-time. One program offloads work to another to be done later.

When you send a message through Slack, it will be stored in a database until the receiver(s) can check it out. That's the case for any form of asynchronous communication; messages need to be stored in a place until a receiver is available to read them.

The simplest form of storing several items in PHP is arrays. So let's see how a queue would look when its messages are stored in an array:

```
$queue = [
    'Send mail to user #1',
    'Send mail to user #2'
];
```

This queue contains two messages. We can *enqueue* a new message, and it'll be added to the end of the queue:

```
enqueue('Send mail to user #3');

$queue = [
    'Send mail to user #1',
```

```
    'Send mail to user #2',
    'Send mail to user #3'
];
```

We can also *dequeue* a message, and it'll be removed from the beginning of the queue:

```
$message = dequeue(); // == Send mail to user #1

$queue = [
    'Send mail to user #2',
    'Send mail to user #3'
];
```

> **Notice:** If you ever heard the term "first-in-first-out (FIFO)" and didn't understand what it means, now you know it means the first message in the queue is the first message that gets processed.

Now a message is a call-to-action trigger; its body contains a string, the receiver interprets this string, and the call to action is extracted. In a queue system, the receiver is called a *worker*. Which is a computer program that constantly dequeues messages from a queue, extracts the call-to-action, and executes it.

Figure 1-5 shows a logical view of how the queue system works. Web application instances enqueue jobs in the queue store while workers keep checking it for messages. Once a worker detects a message is present, it dequeues and processes that message.

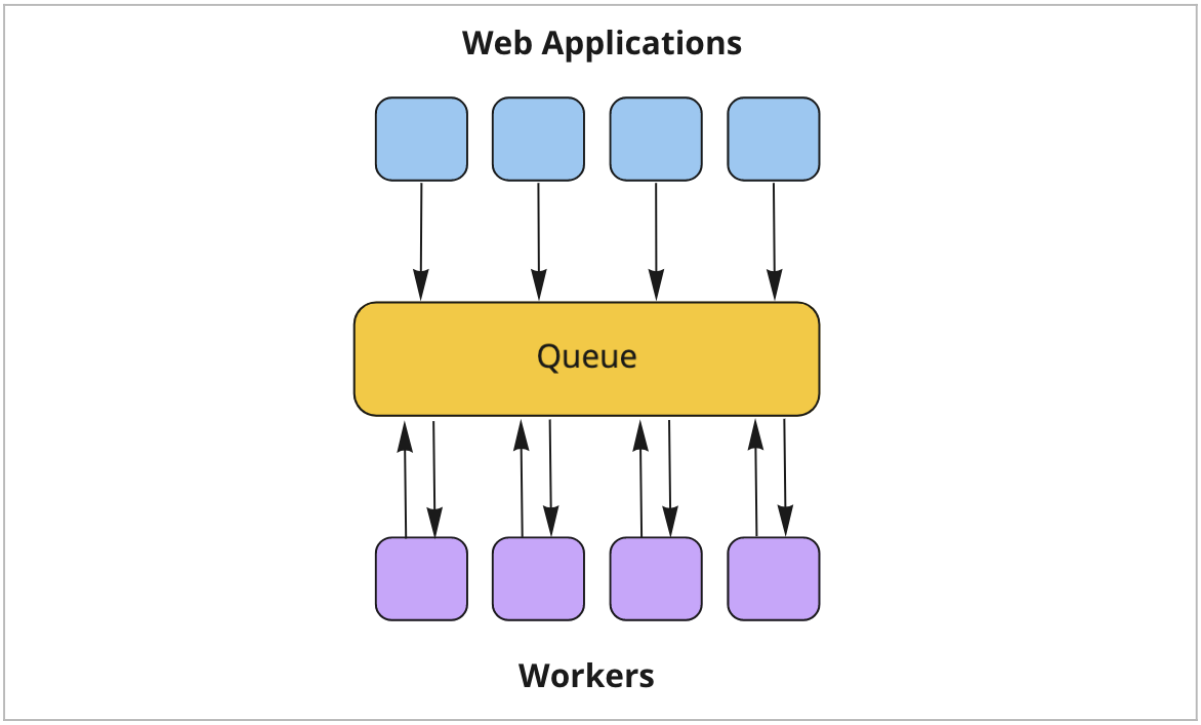

Figure 1-5. Workers in action

As you can see, a web application doesn't wait for the message to be processed; it's one-way communication. It just sends it to the queue and continues handling more client requests. On the other hand, workers don't care where the message is coming from. Their only job is to process a message and move to the next.

Queues in Laravel

Laravel ships with a powerful queue system right out of the box. It supports multiple drivers for storing messages:

- Database
- Beanstalkd
- Redis
- Amazon SQS

Enqueuing messages in Laravel can be done in several ways, and the most basic method is using the `Queue` facade:

```
use Illuminate\Support\Facades\Queue;

Queue::pushRaw('Send mail to user #1');
```

If you're using the database queue driver, calling `pushRaw()` will add a new row in the `jobs`

table with the message "Send mail to user #1" stored in the `payload` field.

Enqueuing raw string messages like this is not very useful. Workers don't know how to interpret plain English and extract the action that needs to be triggered. For that reason, Laravel allows us to enqueue class instances:

```
use Illuminate\Support\Facades\Queue;
use App\Jobs\SendWelcomeEmail;

Queue::push(
    new SendWelcomeEmail(1)
);
```

> **Notice:** Laravel uses the term "push" instead of "enqueue", and "pop" instead of "dequeue".

When you enqueue an object, Laravel will serialize it and build a string payload for the message body. When workers dequeue that message later, they will be able to extract the object and call the proper method to trigger the action.

Laravel refers to a message that contains a class instance as a "Job". To create a new job in your application, you may run this artisan command:

```
php artisan make:job SendWelcomeEmail
```

This command will create a `SendWelcomeEmail` job inside the `app/Jobs` directory. That job will look like this:

```
namespace App\Jobs;

class SendWelcomeEmail implements ShouldQueue
{
    use Dispatchable;
    use InteractsWithQueue;
    use Queueable;
    use SerializesModels;

    public function __construct()
    {
```

```
    }

    public function handle()
    {
        // Execute the job logic.
    }
}
```

When a worker dequeues this job, it will execute the `handle()` method. Inside that method, you should put all your business logic.

> **Notice:** Starting Laravel 8.0, you can use `__invoke` instead of `handle` for the method name.

Another way of sending messages to the queue that workers can interpret is by using a closure (an anonymous function):

```
use Illuminate\Support\Facades\Queue;

Queue::push(function() use ($user){
    Mail::to($user)->send(new Welcome());
});
```

You can even use arrow functions:

```
use Illuminate\Support\Facades\Queue;

Queue::push(
    fn() => Mail::to($user)->send(new Welcome())
);
```

Behind the scenes, Laravel will serialize the closure by converting it into a string form. Then, when the worker picks it up, it can convert it back to its closure form and invoke it.

The Command Bus

Laravel ships with a command bus that can be used to dispatch jobs to the queue. Dispatching through the command bus allows us to use several functionalities I will show

you later.

Throughout this book, we will use the command bus to dispatch our jobs instead of the `Queue::push()` method.

Here's an example of using the `dispatch()` helper, which uses the command bus under the hood:

```
use App\Jobs\SendWelcomeEmail;

dispatch(
    new SendWelcomeEmail(1)
);
```

Or you can use the `Bus` facade:

```
use Illuminate\Support\Facades\Bus;
use App\Jobs\SendWelcomeEmail;

Bus::dispatch(
    new SendWelcomeEmail(1)
);
```

You can also use the `dispatch()` static method on the job class:

```
use App\Jobs\SendWelcomeEmail;

SendWelcomeEmail::dispatch(1);
```

> **Notice:** Arguments passed to the static `dispatch()` method will be transferred to the job instance automatically.

Starting A Worker

To start a worker, you need to run the following artisan command:

```
php artisan queue:work
```

This command will run a program on your machine which will bootstrap an instance of the Laravel application and keep checking the queue for jobs to process.

```
$app = require_once __DIR__.'/bootstrap/app.php';

while (true) {
    $job = $app->dequeue();

    $app->process($job);
}
```

Similar to the HTTP (web application) pipeline, a worker can only handle a single job at any given time. To dequeue jobs faster, you'll need to start multiple workers to process jobs in parallel. We will look into managing workers later in this book.

Now the `queue:work` program that ships with Laravel loads a single instance of your application and uses it to process all jobs. That's different from how a standard web request is handled. Handling web requests in PHP is done by bootstrapping a fresh application instance exclusively for each request and terminating that instance after the request is handled (Figure 1-6).

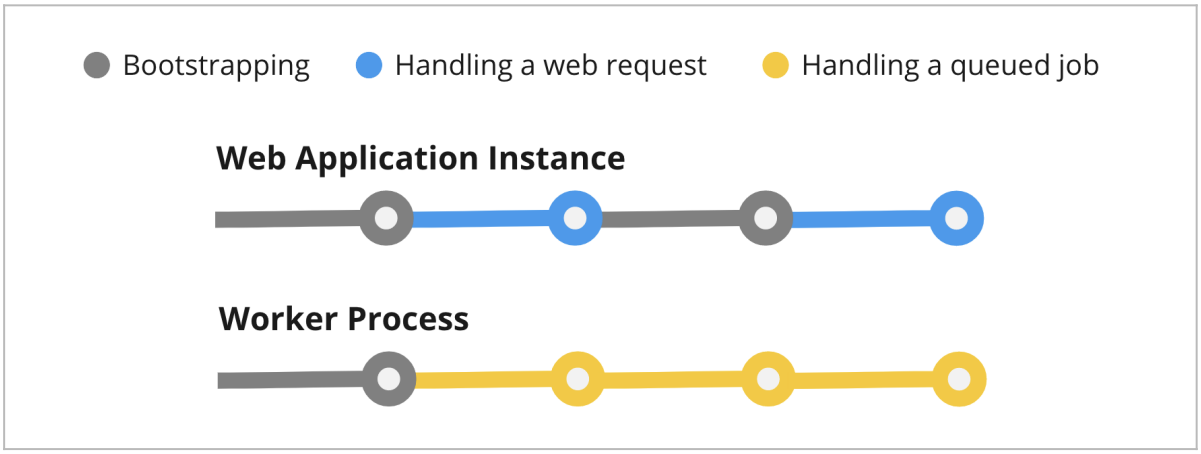

Figure 1-6. Request handling vs. job handling

Bootstrapping an application is a stage with a high resistance coefficient. Therefore, doing it only once in the lifetime of a queue worker has a considerable performance benefit as it will allow the worker to handle more jobs per a given time interval.

However, this benefit comes at a cost. When the same application instance is used, it becomes the software engineer's job to take care of managing the memory. A memory leak somewhere in the code will cause the process to consume all available memory in the

machine and eventually crash it entirely. That's why PHP starts a fresh instance to handle a web request. To free engineers from managing memory.

> **Notice:** Laravel knows the cost of bootstrapping the application for every web request. That's why it gives engineers the option to use Laravel Octane. Web requests are handled by Octane similar to how queued jobs are handled by a queue worker, using the same application instance.

Memory management may seem like a highly complex topic, and it is indeed if you are new to the concept. Luckily, Laravel puts some safeguards in place to prevent workers from eating all machine memory. And in the following chapters of this book, we will discuss the topic in detail and go through the best practices to ensure memory consumption is kept in check.

Another side effect of bootstrapping the application once is that data may leak between jobs if you are not careful. This will cause bugs that are very hard to debug as they will only happen if a worker dequeues jobs that come in a specific order, which is very hard to replicate. We will talk about that in detail in the upcoming chapters as well.

The Advantages of Using Queues

Every virtual private server, aka VPS, has a limit for the concurrent number of requests it can handle. Even serverless offerings like AWS Lambda has this limit. Therefore, you need to handle requests faster to increase the throughput of your VPS or Lambda function without scaling your infrastructure. We've discussed this earlier and mentioned that moving slow stages to the queue will help us handle requests faster. Responding faster to the user is also a significant enhancement to the user experience.

But faster request handling is not the only reason for using queues. Another significant benefit of using queues is decoupling different stages of your application logic. Let's take a look at an example:

```
public function store(Request $request)
{
    $validated = $request->validate(...);

    $order = Order::create($validated);

    Webhooks::each(function($webhook) {
```

```
        $webhook->send($order);
    });

    return OrderResource::make($order);
}
```

In this example, the request goes through several stages:

1. Validation stage.
2. Storage stage.
3. Webhook sending stage.
4. Transformation stage.

The validation, storage, and transformation stages are essential for handling this web request. However, can we say the same about the webhook sending stage? The answer is no. We can definitely send the webhooks later after responding to the user.

Sending those webhooks while handling the request could be a swift operation, and removing this stage won't affect the throughput or user experience that much. But, What if one of the webhook receivers went down temporarily? Should we abort the entire request because one receiver failed? Or should we send that failed webhook again? If we retry, how many times? If we abort the request and throw an error, what happens when the user sends the same request again? Some receivers have already received the webhook; are we going to send them the same webhook again?

There are a lot of questions here. And the simple answer is decoupling. In a loosely coupled application, different parts can be handled in various manners without affecting other parts. Decoupling the webhook sending stage from the request handling process allows us to monitor each webhook sent and retry when needed.

Here's how we can rewrite this controller action:

```
public function store(Request $request)
{
    $validated = $request->validate(...);

    $order = Order::create($validated);

    Webhooks::each(function($webhook) {
        SendWebhook::dispatch($webhook, $order);
    });

    return OrderResource::make($order);
```

```
}
```

We've swapped the actual webhook sending with dispatching a job that will handle sending the webhook later. Inside the queue, we can configure the job for each webhook to automatically retry several times if the receiver fails. We can also configure how much time we should wait before every retry. We will learn how to do that later in this book.

Concurrency

To efficiently deliver all webhooks as soon as possible, we can utilize the concurrency offered by Laravel Octane. However, this comes with the task of memory management since the requests will be handled in the same memory space as we explained earlier in this chapter.

Using Laravel queues, we can achieve concurrency by running multiple workers to process jobs from the queue. That way, you can leave the request handling process of the application to run in the usual manner of handling each request via a separate process without sharing memory.

Rate Limiting

Another benefit we get from decoupling is that we can control the rate a particular part of the application is executed. In the example above, we can control the rate at which webhook sending jobs are executed so we don't hit the receiver's rate limit. When integrating with a third-party API that we have no control over, this is crucial.

Laravel's rate limiting capabilities allow us to easily control the number of times a particular job is executed and the number of times several instances of a job can be executed concurrently. We'll learn about that in the upcoming chapters.

Scheduling

Sometimes you may wish to delay a stage in your web application to be performed after a few seconds. For example, you may want to delay delivering a webhook to allow your database replica instances to read newly written records before a third-party application can read them. To achieve that, you may do something like this:

```
$order = Order::create($validated);

sleep(2);
```

```
$webhook->send($order);
```

The `sleep(2)` stage here will block the execution of the script for 2 seconds before moving to the webhook sending stage. This is obviously something you'll want to avoid in a web application.

Using queues, you can schedule a specific job to be executed after a specific period:

```
$order = Order::create($validated);

SendWebhook::dispatch($webhook, $order)->delay(2);
```

Using `delay(2)` while dispatching the job to the queue instructs Laravel to delay processing this job for 2 seconds. In other words, workers will skip this job for 2 seconds before picking it up. During those 2 seconds, the worker will pick up other jobs in the queue so it won't remain idle.

Prioritization

A computer system is limited by the number of tasks of a particular type it can handle during a given period. Take sending notification emails, for example; if your system can send 1000 emails per hour, you'd want to ensure the most important notifications are sent while less important ones are delayed for later.

Using a queue system, you can give a particular queue a higher priority than others. Workers will always process jobs from that queue before all other queues.

CHAPTER 2

Cookbook

In chapter 1, we used the concepts of industrial pipelines to explain how a Laravel application handles web requests. And we learned how to increase the request throughput by removing some stages from the web pipeline to the queue pipeline.

In this chapter, we will walk through several real-world challenges with solutions that run in production. I met these challenges while building products at Laravel and Foodics, and others collected while supporting the framework users for the past six years.

Before we can jump into these challenges, let's learn how to configure the queue in a fresh Laravel application.

Configuring the Queue

Queued jobs need to be stored somewhere, and Laravel ships with several storage drivers. Each of these drivers has its pros and cons, and we will discover those in detail later in this book. However, the database driver will be the easiest to configure on a local development machine. That's why we'll use it for the examples in this book.

Besides being easy to set up, the database driver gives us high visibility. We can easily open our favorite GUI tool for managing databases and explore the jobs in the queue. We can also delete jobs and make tweaks to test things out.

All we need to get started is to create a database and configure the database connection in the `config/database.php` configuration file.

After that, we need to create the migration that creates the `jobs` database table:

```
php artisan queue:table
```

Running this artisan command will add a migration inside the `database/migrations` directory that creates a `jobs` table. The code inside the migration will look like this:

```
Schema::create('jobs', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('queue')->index();
    $table->longText('payload');
    $table->unsignedTinyInteger('attempts');
    $table->unsignedInteger('reserved_at')->nullable();
    $table->unsignedInteger('available_at');
    $table->unsignedInteger('created_at');
});
```

The `queue` column will hold the name of the queue a job is stored in, and yes, we can dispatch jobs to multiple queues in Laravel. We'll explore the benefits of this later.

The `payload` column will hold the body of the job, which is the string representation of the job object, along with other essential configuration attributes. Like the allowed number of attempts, expiration, timeout, and other ones we will explore later.

The `attempts` column will hold the number of times the job has been attempted.

The `reserved_at` column will hold the timestamp of when a worker picked up this job.

The `available_at` column will hold the timestamp of when workers are allowed to pick this job up. Sometimes it's helpful to delay processing some jobs.

The `created_at` column will hold the timestamp of when the job was dispatched to the queue.

Now that we added the migration and understood what each column is for, let's run the `migrate` command:

```
php artisan migrate
```

If we check the database after this step, we will find the `jobs` table created.

Finally, let's configure Laravel to use the database queue driver. We will edit the `QUEUE_CONNECTION` environment variable inside the `.env` file and set it to `database`:

```
QUEUE_CONNECTION=database
```

Our Laravel application is now ready to store jobs we dispatch from anywhere in the code.

So let's jump into our first challenge.

Sending Email Verification Messages

When it comes to onboarding users to an application, providing the least amount of friction is critical. Unfortunately, some friction is necessary, like making sure the user-provided email address is theirs.

To verify an email address, the application will have to send an email with a link. Then, when the user clicks on that link, we'll know they have access to that email inbox.

The sign-up controller action may look like this:

```
function store()
{
    $this->validate(...);

    $user = User::create([
        ...,
        'email_verified' => false
    ]);

    Mail::to($user)->send(...);

    return redirect('/login?pending_verification=true');
}
```

Here are the stages of this action:

1. Validating the user input.
2. Adding the user to the database with `email_verified = false`.
3. Sending the verification email message.
4. Redirecting the user to the `/login` route.

In the login view, we check the `pending_verification` query parameter and display a message asking the user to check their inbox and verify the email address before they can log in.

Sending the email message might take several seconds as our server will have to communicate with the mailing service provider and wait for a response. The user will have to wait until this communication occurs before they get redirected to the `/login` route and

see a meaningful message.

Forcing the user to wait several seconds during their first interaction with the application isn't the best user experience as it might give them the notion that the application is slow. It'll be great if we can redirect the user to the login screen immediately and send the email message in the background (asynchronously).

Creating and Dispatching the Job

To send the email message via the queue, we will create a `SendVerificationMessage` job:

```
php artisan make:job SendVerificationMessage
```

The job will be created in `app/Jobs/SendVerificationMessage.php`, and will look like this:

```
namespace App\Jobs;

use Illuminate\Contracts\Queue\ShouldQueue;

class SendVerificationMessage implements ShouldQueue
{
    use Illuminate\Foundation\Bus\Dispatchable;
    use Illuminate\Queue\InteractsWithQueue;
    use Illuminate\Bus\Queueable;
    use Illuminate\Queue\SerializesModels;

    public function __construct()
    {
        //
    }

    public function handle()
    {
        //
    }
}
```

To send the email from within the job, we will move the call to `Mail::send()` from the controller action to the `handle()` method of the job:

```
public function __construct(
    private User $user
) {}

function handle()
{
    Mail::to($this->user)->send(...);
}
```

Now back to the controller action, let's dispatch the job using the `dispatch()` static method:

```
function store()
{
    $this->validate(...);

    $user = User::create([
        ...,
        'email_verified' => false
    ]);

    SendVerificationMessage::dispatch($user);

    return redirect('/login?pending_verification=true');
}
```

If we run this controller action, the command bus will send the `SendVerificationMessage` job to the queue.

When we check the `jobs` table, we are will see the following record:

```
{
    "id": 1,
    "queue": "default",
    "payload": "{...}",
    "attempts": 0,
    "reserved_at": null,
    "available_at": 1591425946,
    "created_at": 1591425946
}
```

The `payload` field contains the queued job we just sent, and it's a JSON string that contains information about our job and a serialized version of the `SendVerificationMessage` instance we sent. Here's how it may look:

```
{
   "uuid":"67b31b39-26df-4b39-b2db-c4fb78088fd1",
   "displayName":"App\\Jobs\\SendVerificationMessage",
   "job":"Illuminate\\Queue\\CallQueuedHandler@call",
   "maxTries":null,
   "maxExceptions":null,
   "failOnTimeout":false,
   "backoff":null,
   "timeout":null,
   "retryUntil":null,
   "data":{
      "commandName":"App\\Jobs\\SendVerificationMessage",
"command":"O:32:\"App\\Jobs\\SendVerificationMessage\":1:{s:4:\"user\";O:45:\"Illuminate
\\Contracts\\Database\\ModelIdentifier\":4:{s:5:\"class\";s:15:\"App\\Models\\User\";s:2
:\"id\";i:1;s:9:\"relations\";a:0:{}s:10:\"connection\";s:5:\"mysql\";}}"
   }
}
```

In the following chapters, we'll look closely at each attribute in the job payload. But, for now, notice the value of the `data.command` attribute. This is how the `SendVerificationMessage` job looks after it's serialized.

Starting a Worker

At this point, we managed to dispatch the job to the database queue and redirect the user to the `/login` route so they don't have to wait.

Meanwhile, the job is sitting in the `jobs` database table along with other jobs that resulted from different users signing up.

To start processing those jobs, we need to start a worker:

```
php artisan queue:work
```

Once we run this command, we will see information about the jobs being processed by the worker inside our terminal window:

```
[2020-03-06 06:57:14][1] Processing: App\Jobs\SendVerificationMessage
[2020-03-06 06:57:16][1] Processed:  App\Jobs\SendVerificationMessage
[2020-03-06 06:57:16][2] Processing: App\Jobs\SendVerificationMessage
[2020-03-06 06:57:17][2] Processed:  App\Jobs\SendVerificationMessage
[2020-03-06 06:57:17][3] Processing: App\Jobs\SendVerificationMessage
[2020-03-06 06:57:19][3] Processed:  App\Jobs\SendVerificationMessage
```

The job with `ID = 1` was processed first, then the one with `ID = 2`, then `ID = 3` and so on...

Once the worker picks a job up, it will execute the `handle()` method, which will run our code for sending the email message.

Running Jobs Concurrently

If the queue is filled with verification messages that need to be sent, a new user signing up will have to wait for a long time before they receive their verification message. That message won't be sent until all the previous messages in the queue are processed.

To solve this problem, we will have to process those jobs in parallel.

Each worker can only process a single job at any given time. To process multiple jobs in parallel, we'll need to start more workers:

```
php artisan queue:work
php artisan queue:work
```

> **Notice:** To run multiple workers on our local machine, we must open multiple terminal windows—one for each worker.

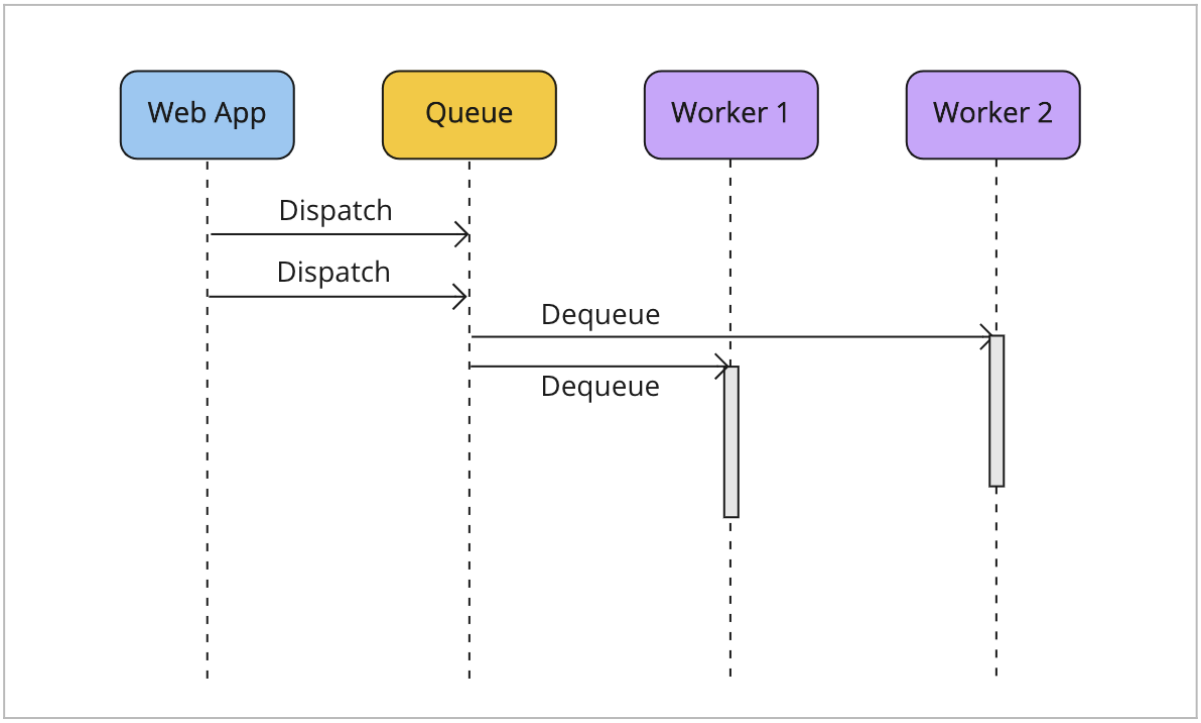

Figure 2-1. Workers processing jobs in parallel

As we can see in this simplified sequence diagram, while the web application dispatches new jobs to the queue, workers are picking up and processing them. Two workers can pick two jobs and process them at the same time. Resulting in consuming the queue faster.

You may think that since the two workers are running in parallel, then there's a chance that both of them will pick up the same job resulting in sending the same verification email to the same user twice. But that's not going to happen. All queue storage drivers use atomic locks to prevent multiple workers from picking the same job.

Let's look at the query executed by the database queue driver when using MySQL:

```
SELECT * FROM `jobs`
WHERE ...
ORDER BY `id` ASC
FOR UPDATE SKIP LOCKED
LIMIT 1
```

The last line in the query has the `FOR UPDATE` instruction, which tells MySQL to take an exclusive lock on the returned row. It also has `SKIP LOCKED`, which tells MySQL to skip any locked row and exclude it from the query. That way, a lock will be put on every job dequeued by a worker. When another worker dequeues, this locked row won't be visible to it.

High Priority Jobs

In the previous challenge, we learned that to process jobs faster, we need to start more workers.

But even with multiple workers running, if the queue is filled with `SendVerificationMessage` jobs and we push a `ProcessPayment` job, the workers won't pick that job up until all previous `SendVerificationMessage` jobs are processed (Figure 2-2).

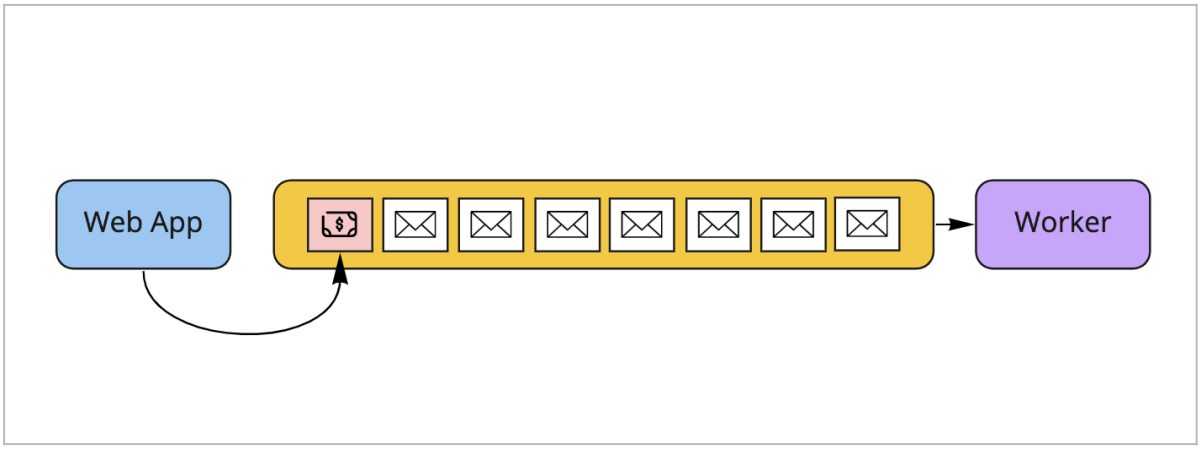

Figure 2-2. The payment job is at the back of the queue

Sending verification messages can wait, but processing payments has a higher priority. So we need to tell our workers to process `ProcessPayment` jobs before others.

Your first thought might be to dispatch payment processing jobs to the beginning of the queue rather than to the back of the queue. That way, workers will always pick these jobs up if they exist before processing any other kinds of jobs (Figure 2-3). However, this will risk having all `SendVerificationMessage` jobs un-consumable by workers if there are always `ProcessPayment` in the queue. So we need something more flexible.

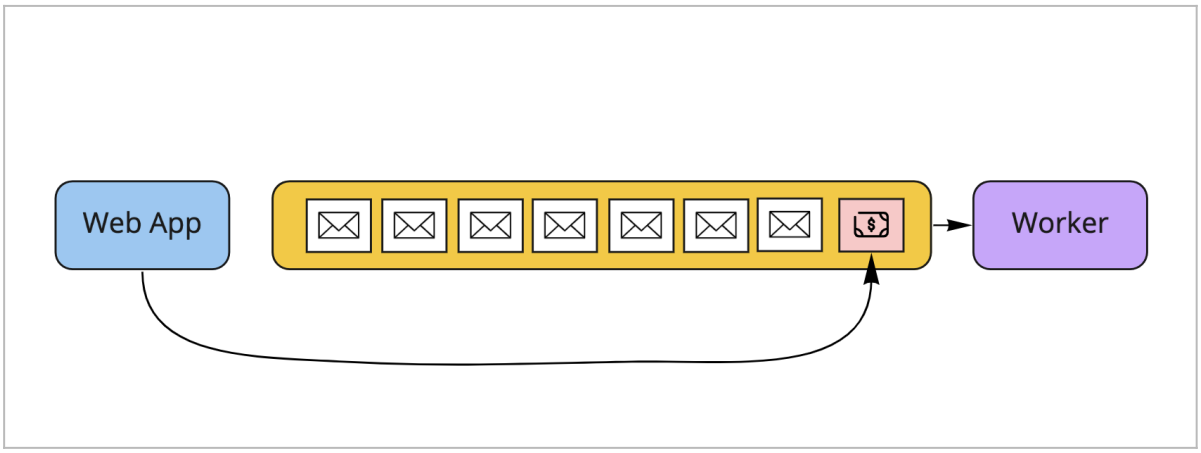

Figure 2-3. Dispatching payment jobs to the beginning of the queue

Laravel allows us to dispatch jobs to multiple queues. So, we can push all `ProcessPayment` jobs to a particular queue, configure our workers to consume jobs from that queue—besides the default queue—and finally give that queue a higher priority.

Dispatching to Multiple Queues

If we take a look at the database queue connection in the `config/queue.php` file, we'll see the following:

```
return [
    'connections' => [
        'database' => [
            'driver' => 'database',
            'table' => 'jobs',
            'queue' => 'default',
            'retry_after' => 90,
        ],
    ]
];
```

Having `queue = default` instructs Laravel to push jobs to a queue named `default` and workers to process jobs from that queue.

To enqueue our `ProcessPayment` jobs in a separate `payments` queue, we'll need to instruct Laravel to do so while dispatching the job explicitly:

```
ProcessPayment::dispatch($payment)->onQueue('payments');
```

Alternatively, we can set a `$queue` public property in the job class and set it to `payments`:

```
namespace App\Jobs;

class ProcessPayment implements ShouldQueue
{
    public $queue = 'payments';
}
```

> **Notice:** By defining the `$queue` public property, we no longer need to call `onQueue()` when dispatching the job. Each time this job is dispatched, Laravel will send it to the `payments` queue automatically.

Processing Jobs from Multiple Queues

At this point, all our `ProcessPayment` jobs are stored in the `payments` queue while all the other jobs are in the `default` queue. And in the previous challenge, we had a few workers running and processing jobs from the `default` queue by default.

Let's also instruct our workers to process jobs from the `payments` queue. We'll do this by terminating those workers and starting new ones with the `--queue` option configured:

```
php artisan queue:work --queue=payments,default
```

This will instruct the worker to pop—or dequeue—jobs from the `payments` and `default` queues, with the `payments` queue having a higher priority (Figure 2-4). Jobs from the `default` queue will not be processed until the `payments` queue is cleared.

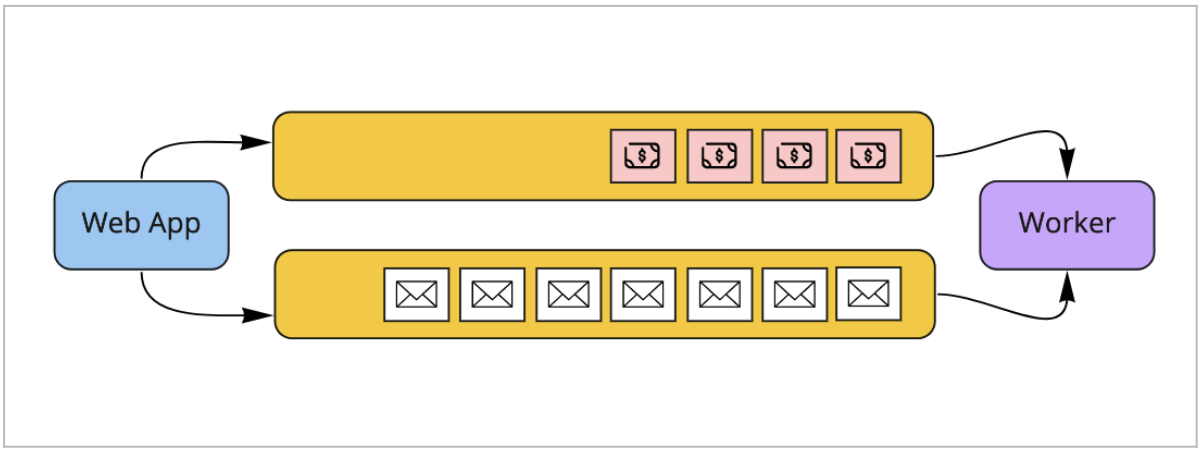

Figure 2-4. Dispatching to multiple queues

Using Separate Workers for Different Queues

Using `--queue=payments,default` in the `queue:work` command allowed us to configure the queue priority for each worker. However, if the `payments` queue is always filled with jobs, jobs from queues with lower priority may not run at all.

In most production cases I've seen, this was never a problem. However, if we want to ensure that there will always be a worker processing jobs from each queue, we can configure a separate worker for each one:

```
php artisan queue:work --queue=payments
php artisan queue:work --queue=default
```

Having these two workers running ensures that there's always a worker processing jobs from the `payments` queue while another worker is processing jobs from the `default` queue.

One thing with this, though, is if the `default` queue is empty while the `payments` queue is full, the second worker will stay idle. So it's better if we instruct the workers to process jobs from the other queue if their main queue is empty.

> **Notice:** Running workers consume memory and CPU cycles. Therefore, we should always try to use all available workers most efficiently.

To do this, we'll start our workers with these configurations:

```
php artisan queue:work --queue=payments,default
```

```
php artisan queue:work --queue=default,payments
```

Now, the first worker will consume jobs from the `payments` queue as long as it has jobs; if not, it'll consume jobs from the `default` queue. The second worker will consume jobs from the `default` queue if it's busy; otherwise, it'll start looking at the `payments` queue.

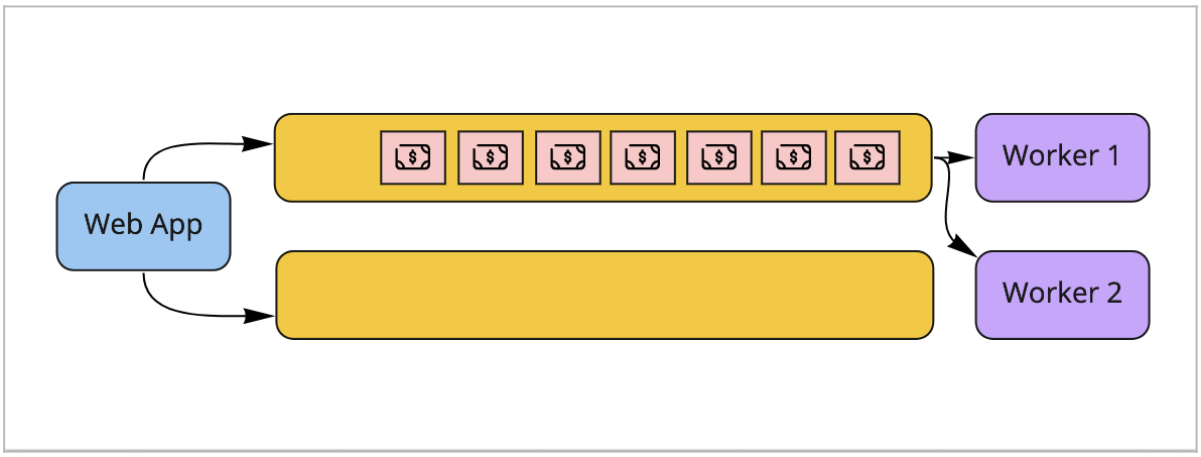

Figure 2-5. Running multiple workers

Figure 2-5 shows how worker two will not remain idle if the `default` queue is empty. Instead, it will start consuming jobs from the busy `payments` queue. No wasted resources.

Retrying Failed Jobs

We're still looking into configuring email verification jobs. The `handle()` method of the `SendVerificationMessage` job currently looks like this:

```
function handle()
{
    Mail::to($this->user)->send(...);
}
```

This simple job's success depends on the network connectivity, availability of the mail service provider, and its acceptance to the mail sending request. Some of these things we have some control over (network connectivity, sending valid payload, paying the bills, etc...), others not (availability of the mail service). In all cases, we need to plan for when a job fails as much as when it succeeds.

When a failure happens inside one of the stages of handling a web request, the application will throw an exception, and the user will get a message indicating the request failed. They

can retry the request later or contact customer support. We can also configure logging to get notified when one of our users faces an error. For queued jobs that run asynchronously, though, handling failure can be a bit tricky. Here's what Laravel does by default when a job fails:

1. Deletes the job from the queue.
2. Reports the failure in the terminal window running the `queue:work` command.
3. Dispatches a `JobFailed` event.
4. Stores the job information in a failed jobs store.

This failed jobs store is known in the computer science world as *Dead-letter Queue*. Its primary purpose is logging and debugging unprocessed jobs. Laravel stores a record of failed jobs in a `failed_jobs` database table. Inside this table, a record may look like this:

```
{
    "id": 300,
    "uuid": "67b31b39-26df-4b39-b2db-c4fb78088fd1",
    "connection": "database",
    "queue": "default",
    "payload": "{...}",
    "exception": "...",
    "failed_at": 1591425946
}
```

Laravel stores the job's UUID, the queue connection the job uses, the queue the job was dispatched to, the payload, the exception that was thrown, and the timestamp of when the job failed.

This makes it easy for us to inspect the exception and decide on what we need to do. For example, if it's a non-transient error, like a validation error, or the provider is out of business, we'll need to change the job or the mail service configuration. On the other hand, if it's a temporary outage, we'll have to retry the job later.

Inspecting the Dead-letter Queue

To inspect the dead-letter queue, Laravel provides us with the `queue:failed` artisan command.

```
php artisan queue:failed
```

This command outputs a table in the terminal window that contains all the failed jobs

records:

```
+--------------+------------+---------+-------------------------+---------------------+
| ID           | Connection | Queue   | Class                   | Failed At           |
+--------------+------------+---------+-------------------------+---------------------+
| d615e9c5s... | database   | default | SendVerificationMessage | 2022-06-11 15:30:07 |
+--------------+------------+---------+-------------------------+---------------------+
```

As we can see, the exception details isn't included in the output of this command. That's mainly for formatting reasons. Displaying the full stacktrace of an exception inside the terminal will not be readable for a human. It is advisable to use a GUI to inspect the `failed_jobs` database table to browse through all the failed jobs and inspect the exception details.

Retrying a Job Manually

When an email verification job fails, the user won't be able to use the application since their email isn't verified. They will not get any feedback that there was an internal failure, so they will keep waiting and waiting until they contact customer support to report the issue. Typically customer support will forward the incident to developers, who, ideally, know about the issue by now and are working on fixing it.

Once the issue is fixed, developers will want to send all those failed jobs in the dead-letter queue. For that reason, Laravel ships with a `queue:retry` artisan command we can use.

Here for example, we can retry a job by providing the UUID to the command:

```
php artisan queue:retry 67b31b39-26df-4b39-b2db-c4fb78088fd1
```

You may think this command processes the job on the spot, but that's not what happens. This command only re-dispatches the job back to the queue, where it will be treated like any newly dispatched job. The only difference is that the job will keep its UUID. New jobs always get a new UUID, but failed jobs keep their UUID.

Here are the steps Laravel takes to retry a job:

1. Queries the `failed_jobs` table.
2. Gets the record.
3. Dispatches the job to the queue.
4. Deletes the record.

Retrying All Failed Jobs

A temporary outage in one of the systems that our application uses can stay for hours, and sometimes days, depending on how big the issue is. I've seen some services go offline for hours which caused a huge backlog of failed jobs in the dead-letter queue.

Retrying a large number of failed jobs manually can be a tremendous amount of work. For that reason, Laravel allows us to retry all of the failed jobs by running a single command:

```
php artisan queue:retry all
```

When that command runs, Laravel will go over all failed jobs and dispatch them to the queue one by one. This, obviously, can take quite some time for a large number of jobs. So if we run this command and it takes some time to return, don't panic!

Keeping the Dead-letter Queue in Check

As we mentioned, the dead-letter queue can be filled with jobs when things go wrong. However, for some non-transient failures, a retry won't help. For those cases, Laravel ships with two commands that can help us keep the dead-letter queue in check.

The first is the `queue:prune-failed` command:

```
php artisan queue:prune-failed
```

When this command runs, it will delete failed jobs older than 24 hours. We can control the number of hours to retain failed jobs by providing a `hours` option:

```
php artisan queue:prune-failed --hours=48
```

The other command is the `queue:flush` command. This one flushes the dead-letter queue and deletes all failed jobs:

```
php artisan queue:flush
```

Configuring Automatic Retries

Retrying jobs manually is excellent, but it's not ideal. Most of the work we'd prefer to do asynchronously involves communicating with remote services, in which transient failures are expected. These failures include loss of connectivity, unavailability of a service, declines due to rate limiting, or timeouts. However, they are all self-correcting, and if the same job is retried after a bit, it is likely to succeed.

Laravel allows us to configure a retry strategy for our jobs. It can be as simple as instructing workers to retry every job several times before considering it a failure. To do that, we may start our workers with the `--tries` option:

```
php artisan queue:work --tries=3
```

This worker will attempt every job 3 times–if it didn't finish–before considering it a failure. Behind the scenes, Laravel will release it back to the queue so workers can pick it up again. You can think of releasing a job back to the queue as pushing a new job.

> **Notice:** Every time a worker picks a job up, it increments an internal "attempts" counter. That's how it keeps track of how many times the job has been retried.

Figure 2-6 shows the three stages of a job that got retried. First, the job was pushed to the queue by the web app. Then, the worker picks it up. Finally, the job was released back to the queue when it failed to complete.

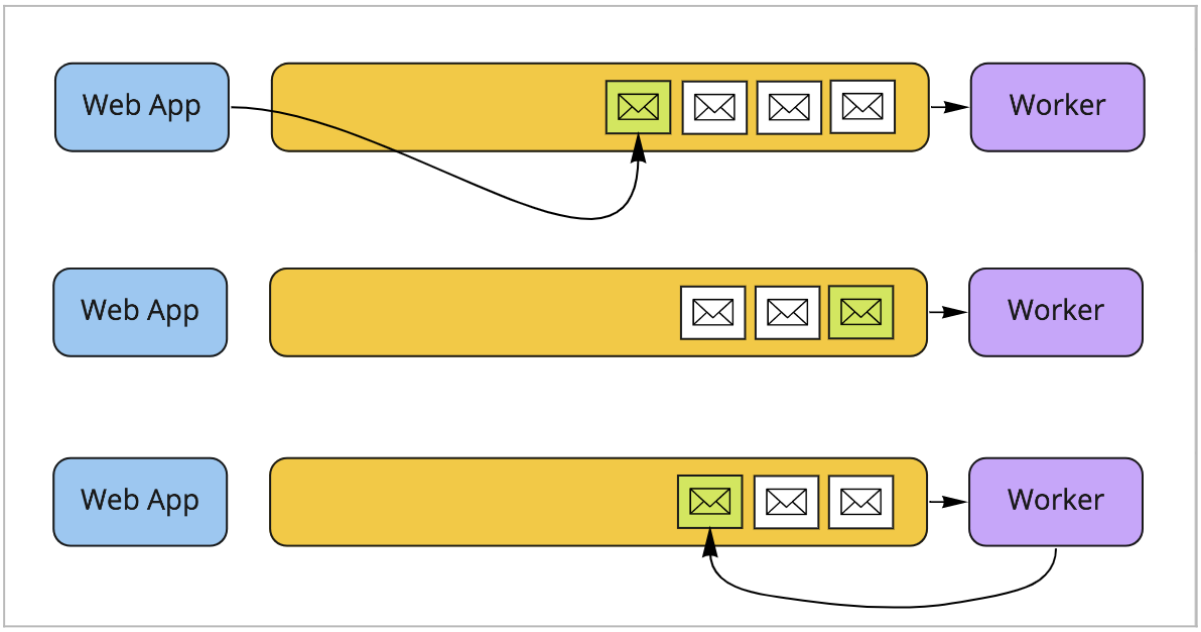

Figure 2-6. Releasing a job back to the queue.

If the job fails again, it will be released back to the queue again. Finally–if it fails another time–the worker will move it to the dead-letter queue. Because by this time, the job would have been tried three times.

You should notice in Figure 2-6 that a released job may not be retried immediately by the worker if there are other jobs in the queue. Also, if multiple workers are consuming the queue, different workers may make attempts. There's no guarantee that a job will be retried immediately nor the same worker will retry it.

Job-Specific Tries

Not all jobs should be retried the same number of times. It's important we put in place a retry strategy that is tailored for each specific case. If we don't retry a job enough, it will fail and require manual intervention. If we retry too much, we would be wasting valuable computing and networking resources.

Laravel allows us to configure the number of tries on a job level by adding a `$tries` public property to the class:

```
namespace App\Jobs;

class SendVerificationMessage implements ShouldQueue
{
    public $tries = 3;
}
```

Retrying a `SendVerificationMessage` job three times seems appropriate. But we may find it approriate to retry a `ProcessPayment` ten times.

```
namespace App\Jobs;

class ProcessPayment implements ShouldQueue
{
    public $tries = 10;
}
```

Whatever value we use in the `$tries` public property will override the value specified in the `--tries` worker option. With that in mind, we can omit the `--tries` option from the worker command and let it fail jobs on the first attempt and only configure specific jobs to retry multiple times.

Delaying Tries

If a job fails to complete and multiple tries were configured, the worker will dispatch the job back to the queue. That newly dispatched job will be picked up again by a worker. This could happen in a few minutes, or a few milliseconds, depending on how busy the queue is and the number of workers consuming jobs from that queue.

The default retry policy that ships with Laravel ensures that a retry would happen as soon as possible (once a worker is free to pick the job up). In many cases, we may want to control the interval between retries to give the service enough time to recover from a transient failure. To do that, we may add a `$backoff` public property in the job class:

```
namespace App\Jobs;

class SendVerificationMessage implements ShouldQueue
{
    public $tries = 3;
    public $backoff = 20;
}
```

By doing this, we're instructing our workers to retry this job after 20 seconds on each failure.

Figure 2-7 shows a job that got picked up at 00:00 and ran for 10 seconds but didn't complete, the job was released back to the queue and the worker picked it up again after

the 20 second backoff passed at 00:30. This time the job ran for 15 seconds and didn't complete, so the worker released it back to the queue at 00:45. The worker waited for another 20 seconds to pick it up again at 01:01.

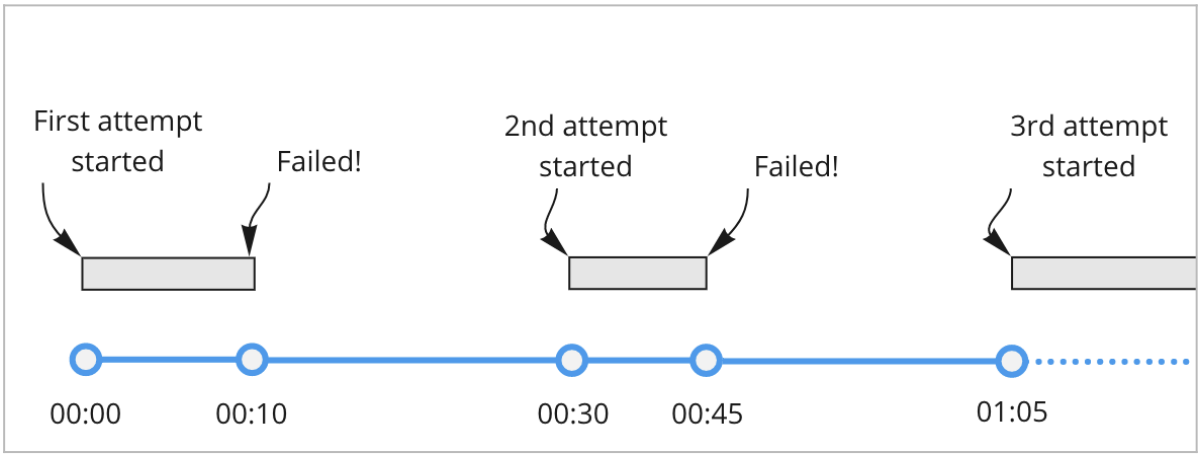

Figure 2-7. Retrying timeline.

Keep in mind that the backoff time only makes the job invisible to workers during the interval, it doesn't guarantee the job will be retried exactly after the backoff period passes. Figure 2-7 shows a unique case in which the worker was not occupied by other work when the backoff period passed, so it was able to pick the job up immediately. I wanted to stress this point enough because it's a widespread misconception that causes a lot of confusion when people think a released job will run directly after the backoff period.

Exponential Backoff

In our example, waiting 20 seconds for the transient failure to resolve seems enough. But what if the job fails again? That would probably mean the transient failure is a long-term one. Retrying every 20 seconds can be a waste of resources in that case. It might be a good idea to increase the delay after each attempt:

```
namespace App\Jobs;

class SendVerificationMessage implements ShouldQueue
{
    public $tries = 3;
    public $backoff = [20, 60];
}
```

Using an exponential backoff, we instructed the worker to delay retrying the job 20 seconds the first time, but then delay the third retry 60 seconds. Giving the service more time to recover and saving our system resources for other tasks in the meantime.

> **Notice:** If we have `$tries = 4`, the fourth attempt will be delayed 60 seconds as well.

We can also configure backoff on a worker level:

```
php artisan queue:work --backoff=20,60
```

This will delay attempting all failing jobs processed by this worker for 20 seconds after the first attempt and 60 seconds for each attempt as it advances.

Not Really FIFO

You may have figured this out already, but dispatching jobs to a queue in a specific order doesn't mean they will be processed in that exact order. The workers will pick them up in the exact order, but the processing may not happen in the same order because some jobs may fail and be returned to the queue and retried later.

Imagine two `SendVerificationMessage` jobs were dispatched for two users (Figure 2-8), let's call them Zain and Adam. Adam signed up before Zain so his verification job got dispatched first. By the time the worker picked Adam's job up, the mail service provider was down, so it got released back to the queue, and the worker moved to process Zain's job. Luckily, the service was back online, and the job was processed. Then after the backoff period passed, Zain's job got picked up by the worker and was processed successfully.

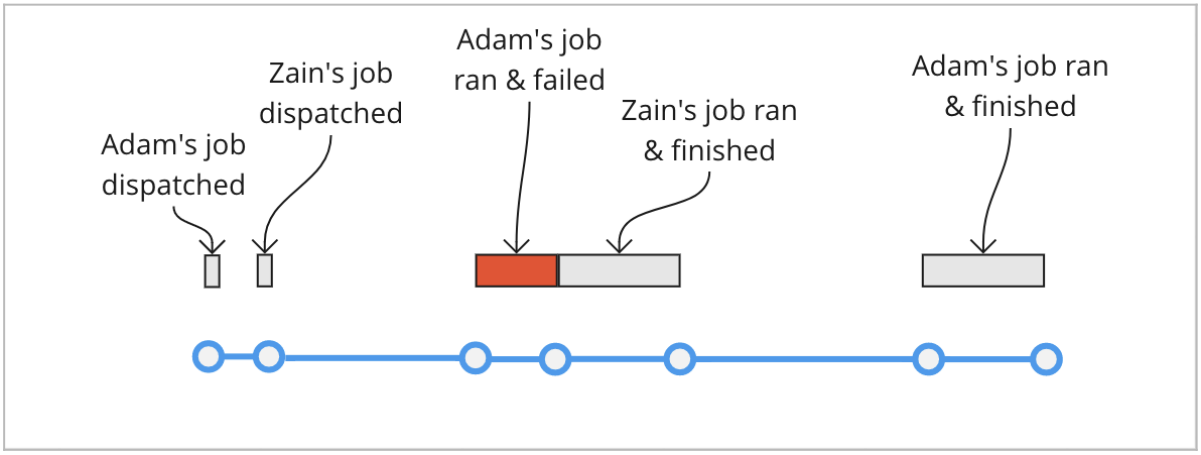

Figure 2-8. Jobs are picked up in exact order but processed in a different order.

There is a way to instruct Laravel to process a chain of jobs in exact order and wait for one job to finish successfully before moving to the next. We will look into that in a future

chapter. But the general rule is that we shouldn't expect jobs to be finished in the same order they were dispatched. It is another common misconception that I'm hoping to clear up in this book.

Canceling Abandoned Orders

Now let's move to a different challenge: we'll look into a familiar pattern used in eCommerce. When users add items to their shopping cart and start the checkout process, you want to reserve these items for them. However, if a user abandoned an order—they never canceled or checked out—you will want to release the reserved items back into stock so other people can order them.

To do this, we will schedule a job once a user starts the checkout process. This job would check the order status after **one hour** and cancel it automatically if it wasn't completed by then.

Delay Processing a Job

Let's see how such a job can be dispatched from the controller action:

```
class CheckoutController
{
    public function store()
    {
        $order = Order::create([
            'status' => Order::PENDING,
            // ...
        ]);

        MonitorPendingOrder::dispatch($order)
            ->delay(3600);
    }
}
```

The first procedure is creating an order entry and setting the status to `pending`. After that, we dispatch the job and chain a `delay(3600)` method after `dispatch()`. This method instructs Laravel to delay processing the `MonitorPendingOrder` for 3600 seconds (1 hour).

We can also set the delay using a `DateTimeInterface` implementation:

```
MonitorPendingOrder::dispatch($order)->delay(
    now()->addHour()
);
```

Dispatching jobs with a delay works similarly to releasing jobs that didn't complete back to the queue with a backoff. During the delay period, the job will be invisible to workers.

> **Warning:** Using the SQS driver, you can only delay a job for 15 minutes.

Here's a quick look inside the `handle()` method of that job:

```
public function handle()
{
    if ($this->order->status == Order::CONFIRMED ||
        $this->order->status == Order::CANCELED) {
        return;
    }

    $this->order->markAsCanceled();
}
```

When the job runs—after an hour—we'll check if the order was canceled or confirmed and return from the `handle()` method. Otherwise, we'll cancel the order if it is still pending.

Sending Users a Reminder Before Canceling

Now let's enhance the user experience and try to increase conversion by sending the user an SMS notification to remind them about their order before canceling it. To do this, we will send an SMS message every 15 minutes until the user completes the checkout, or we cancel the order after 1 hour.

First, let's adjust the delay on the job processing to be 15 minutes instead of an hour:

```
MonitorPendingOrder::dispatch($order)->delay(
    now()->addMinutes(15)
);
```

Now when this job runs, we want to check if an hour has passed and cancel the order. Or, if

we're still within the hour, we'll send an SMS reminder and release the job back to the queue with a 15-minute delay.

```
public function handle()
{
    // Check the status of the order
    if ($this->order->status == Order::CONFIRMED ||
        $this->order->status == Order::CANCELED) {
        return;
    }

    // Cancel the order if it's older than 59 minutes
    if ($this->order->olderThan(59, 'minutes')) {
        $this->order->markAsCanceled();

        return;
    }

    // Send a reminder SMS message
    SMS::send(...);

    // Release the job back to the queue with a delay
    $this->release(
        now()->addMinutes(15)
    );
}
```

Calling `$this->release()` inside a job instructs the worker to put the job back in the queue to be retried later. This method accepts an optional argument representing the delay we want to set. In our case, we're delaying the processing of the released job for 15 minutes.

Ensuring the Job Has Enough Attempts

Every time the worker picks up the job, it'll increment the attempts counter. And given that jobs are allowed to be attempted only a single time by default, we need to make sure our job has enough `$tries` to run four times:

```
class MonitorPendingOrder implements ShouldQueue
{
    public $tries = 4;
}
```

Assuming the job was dispatched at `00:00` and that the worker was available to pick it up when the delay period expires, the time for the job will be like this (Figure 2-9):

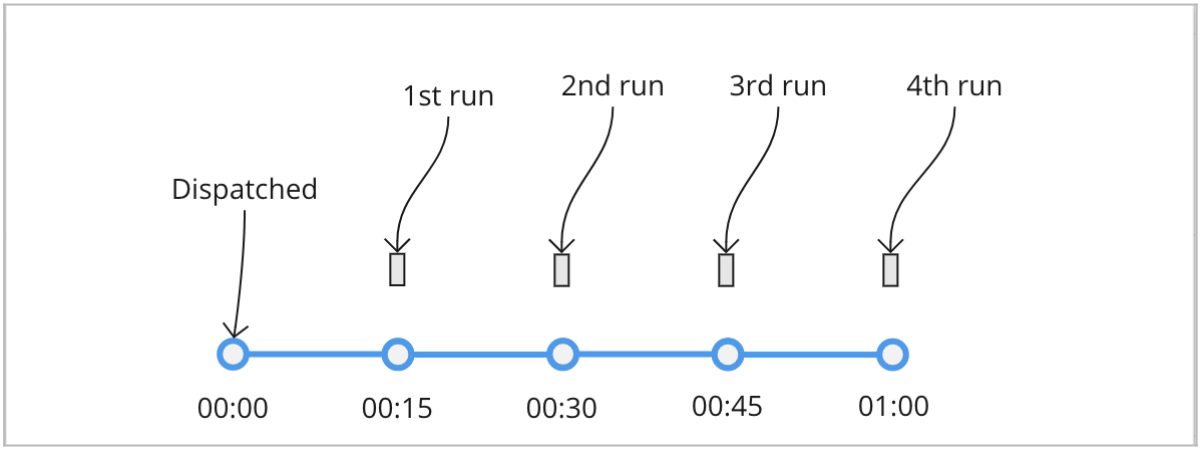

Figure 2-9. The job is attempted four times.

If the user confirms or cancels the order, say after 20 minutes, the job will be deleted from the queue when it runs on the attempt at 30 minutes, and no SMS will be sent.

This is because we have this check at the beginning of the `handle()` method:

```
if ($this->order->status == Order::CONFIRMED ||
    $this->order->status == Order::CANCELED) {
    return;
}
```

Delay and Backoff Periods

As we've mentioned, there's no guarantee that workers will pick the job up after the delay or backoff period passes. If the queue is busy and not enough workers are running, the `MonitorPendingOrder` job in this example may not run enough times to send the three SMS reminders before canceling the order (Figure 2-10).

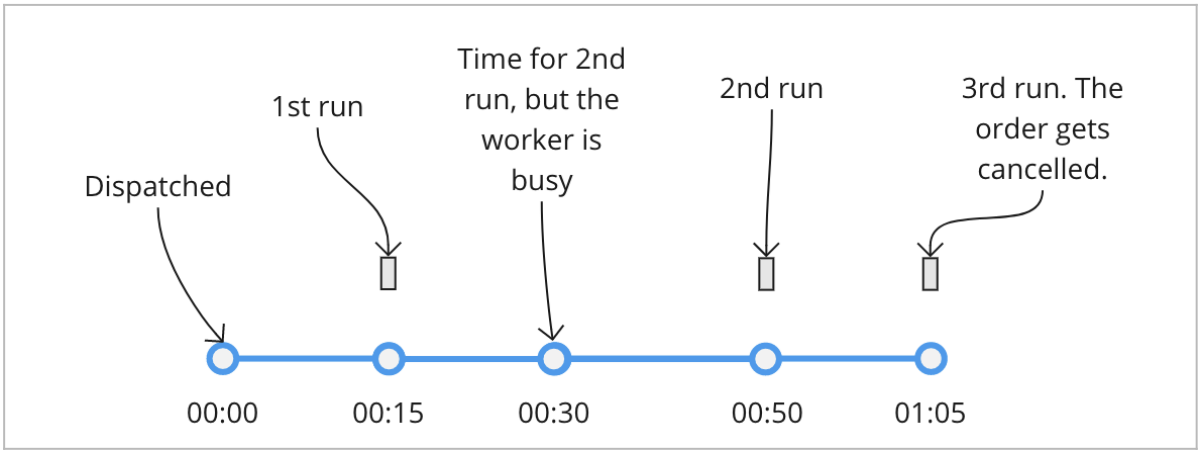

Figure 2-10. The job runs 3 times only.

To increase the chance of your delayed jobs getting processed on time, you need to ensure you have enough workers to empty the queue as quickly as possible. This way, when the job becomes available, a worker will be ready to run it immediately.

Sending Webhooks

Another common use case for queues is handling the task of sending webhooks. Which is a method to extend applications by allowing 3rd party integrations to subscribe to specific events. When any of these events occur, the application calls a webhook URL configured by the integration:

```
SendWebhook::dispatch($integration, $event);
```

Inside the `SendWebhook` job, we're going to use Laravel's built-in HTTP client to send requests to the integration URL:

```
use Illuminate\Support\Facades\Http;

class SendWebhook implements ShouldQueue
{
    // ...

    public function handle()
    {
        $response = Http::post(
            $this->integration->url,
            $this->event->data
        );
```

```
    }
}
```

Now every service that offers webhooks must have a retry policy in place. This policy is usually communicated with developers and partners in the service documentation or integration agreements.

The challenge we have here is implementing an exponential webhook retry policy in which the retry backoff is increased by 15 minutes after every attempt. The service keeps attempting the webhook for 24 hours and sends an alert to the partner at the end of this period.

Dynamic Exponential Backoff

First, let's work on the 15-minute backoff increase part. We know that to configure an exponential backoff, we need to define a `$backoff` property in the job class and provide an array of backoff periods. But since this job will be be retried times, let's set the backoff dynamically instead.

To do that, we will handle the webhook failure inside the `handle` method instead of relying on the worker's exception handler. Inside our handler, we will release the job back to the queue with a delay period defined by the number of the current attempt. Check this out:

```
$response = Http::post(...);

if ($response->failed()) {
    $this->release(
        now()->addMinutes(15 * $this->attempts())
    );
}
```

> **Notice:** The `attempts()` method returns the number of times the job has been picked up. If it's the first run, `attempts()` will return `1`.

What's happening here is that we release the job back to the queue with an increasing delay. Here's how the timeline will look like if the job keeps failing:

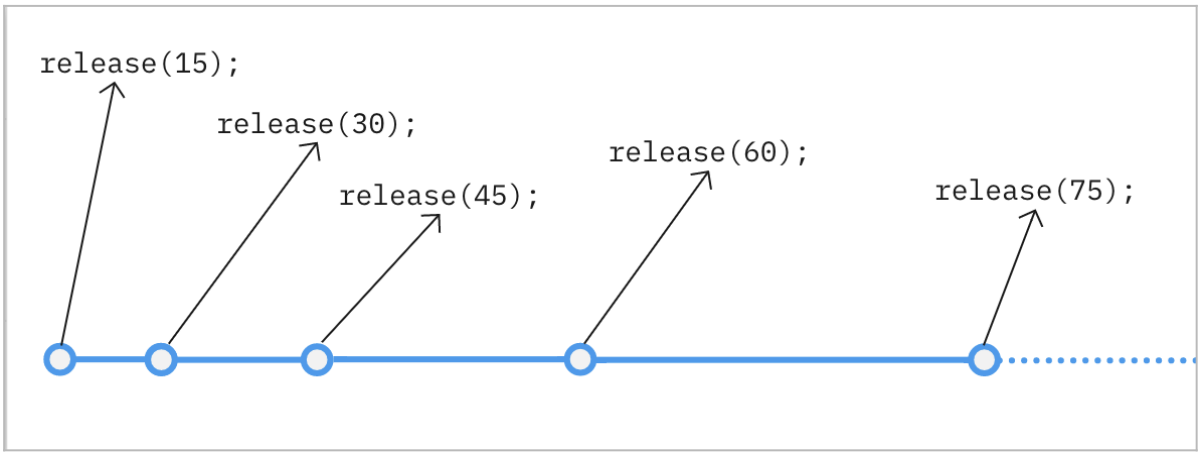

Figure 2-11. Dynamic exponential backoff.

The backoff period between attempts will keep increasing dynamically. We saved a bit of brain power for other stuff and let the computer build the incremental backoff strategy for us.

Configuring Job Expiration

Now let's look into retrying the job for 24 hours. We could do the math and calculate how many attempts a whole day can allow based on our exponential backoff. But there's an easier way:

```
class SendWebhook implements ShouldQueue
{
    public function retryUntil()
    {
        return now()->addDay();
    }
}
```

Adding a `retryUntil()` public method to our job class instructs Laravel to set an expiration date for the job. In this case, the job will expire after 24 hours.

> **Warning:** The expiration is calculated by calling the `retryUntil()` method when the job is first dispatched. If you release the job back, the expiration will not be re-calculated.

Now the worker will fail this job if it was attempted after the 24-hour period. But since all jobs by default are allowed only one attempt, we need to configure this job to retry an

unlimited number of times. That way, the worker will keep retrying it until it expires. To do this, we need to set the `$tries` public property to `0`:

```
class SendWebhook implements ShouldQueue
{
    public $tries = 0;
}
```

Handling Job Failure

Now that we implemented the incremental backoff strategy and configured the job expiration. It's time we work on the last part of our webhook retry policy, which is sending an alert to the partner if a webhook keeps failing for over 24 hours.

To do this, we're going to add a `failed()` public method to our job class:

```
class SendWebhook implements ShouldQueue
{
    // ...

    public function failed(Exception $e)
    {
        Mail::to(
            $this->integration->developer_email
        )->send(...);
    }
}
```

When a job exceeds the assigned number of attempts or reaches the expiration time, the worker will call this `failed()` method and pass a `MaxAttemptsExceededException` exception. Inside this method, we will use the `Mail` facade to send an email to the integration developer with details of the failing webhook.

We've discussed in a previous challenge that sending mail is an error-prone process as it can face transient failures. Therefore, we need to have a retry policy in place to handle these failures. If the mail sending fails inside the `failed` method of the `SendWebhook` job, it will not be retried automatically.

We need to isolate this task into its job so we can configure retries and exponential backoff. So let's implement a `SendWebhookFailedWarning` job in our project:

```
class SendWebhookFailedWarning implements ShouldQueue
{
    public $tries = 3;
    public $backoff = [20, 60];

    public function __construct(
        private Integration $integration
    ) {}

    public function handle()
    {
        Mail::to(
            $this->integration->developer_email
        )->send(...);
    }
}
```

Inside the `failed` method of the `SendWebhook` job, we can dispatch this new job:

```
class SendWebhook implements ShouldQueue
{
    public function failed(Exception $e)
    {
        SendWebhookFailedWarning::dispatch($this->integration);
    }
}
```

Provisioning a Forge Server

In the previous challenges, we learned we could control the lifetime of a job by limiting retries or setting a job expiration. At this point, you may think that using expiration is a much better option for all use cases. However, in this challenge, we'll see that using expiration is a bit risky and that you might need to think twice before using it.

In this challenge, we'll look into using Laravel Forge to provision a new server. We need to communicate with the Forge API by sending a POST request to the `/servers` endpoint. The request might take a few seconds, and we'll receive the server ID in the response. But that doesn't mean the server is ready to go.

After Forge responds to our request, it will continue provisioning the server, which may take a few minutes. So, we'll need to keep checking the server status and update our local

storage when the server is ready.

Here's how the `store()` controller action in which we create the server may look:

```
public function store()
{
    $server = Server::create([
        'is_ready' => false,
        'forge_server_id' => null
    ]);

    ProvisionServer::dispatch($server, request('server_payload'));
}
```

First, we create a record in our database to store the server and set `is_ready = false` to indicate that this server is not usable yet. Then, we dispatch a `ProvisionServer` job to the queue.

Here's how the job may look:

```
class ProvisionServer implements ShouldQueue
{
    public function __construct(
        private Server $server,
        private array $payload
    ) {}

    public function handle()
    {
        if (! $this->server->forge_server_id) {
            $response = Http::timeout(5)->post(
                '.../servers', $this->payload
            )->throw()->json();

            $this->server->update([
                'forge_server_id' => $response['id']
            ]);

            return $this->release(120);
        }

        $response = Http::timeout(5)->get(
            '.../servers/'.$this->server->forge_server_id,
        )->throw()->json();
```

```
        if ($response['status'] == 'provisioning') {
            return $this->release(60);
        }

        $this->server->update([
            'is_ready' => true
        ]);
    }
}
```

Here, we check if a `forge_server_id` is not assigned—which indicates we haven't created the server on Forge yet—and create the server by contacting the Forge API. After that, we update the `forge_server_id` attribute with the ID from the response.

Now that the server is created on Forge, we send the job back to the queue using `release(120)`. This gives the server a couple of minutes to finish provisioning.

When a worker picks that job up again—after two minutes or more—, it's going to check if the server is still provisioning and release the job back to the queue with a 60-second delay.

Finally, we mark the server as ready if Forge is done provisioning it.

Managing Attempts

As we've learned, workers will attempt the job only once by default. If an exception was thrown or the job was released back to the queue, a worker will mark it as failed.

Our challenge here requires that we run the job multiple times to give the server enough time to provision. We could use `retryUntil` so the job keeps retrying for, say, 5 minutes. However, if our workers are busy with too many jobs, they might not be able to pick this job up in time. Here's how this might happen.

In Figure 2-12, the job ran 2 minutes after it was dispatched. The server was created and the job was released back to the queue with a 2-minute delay. However, after 5 minutes from dispatch, the job expires. When a worker became available to pick the job up, it found it expired, so it never ran it.

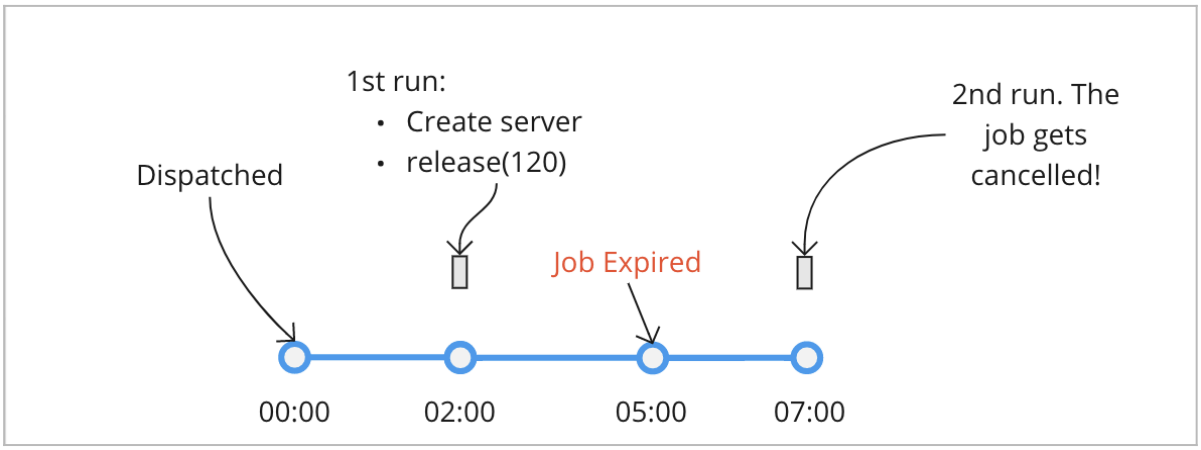

Figure 2-12. Job expires before the server provisions.

> **Warning:** Remember that releasing the job with a delay doesn't mean the job will run exactly after the delay period. It only means workers will ignore the job for that time.

To ensure this doesn't happen, we will rely on attempting the job a specific number of times rather than setting a job expiration. So, if we want to give the server at least 5 minutes to provision, we'll need to attempt the job five times. So let's go ahead and set `$tries` to 5:

```
class ProvisionServer implements ShouldQueue
{
    public $tries = 5;

    // ...
}
```

In Figure 2-13, the job keeps attempting until the server was provisioned 8 minutes after it was dispatched.

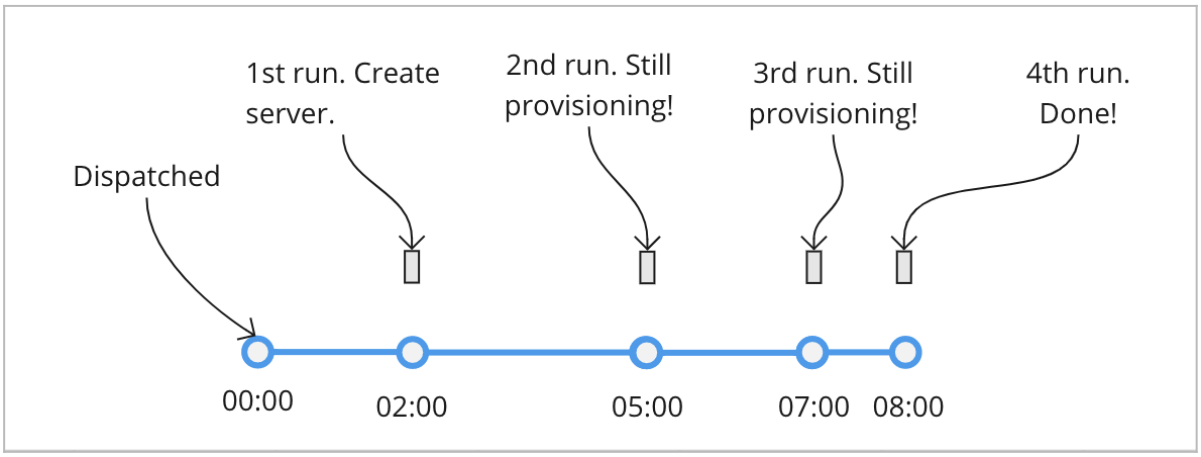

Figure 2-13. Job has enough attempts to give the server 5 minutes.

Limiting Exceptions

Being able to release jobs and attempt them several times allows us to deal with long-running external processes without keeping a worker stuck waiting for them to finish. We release the job, move to handle another, and then get back to it to check the status. However, this means the job will be retried five times if an exception was thrown from inside the `handle()` method.

Exceptions could be thrown due to a temporary outage in the Forge API or a validation error sent from the server creation endpoint. Attempting the job five times is too much in this case, especially if the exception needs our attention, like a validation exception from Forge.

For that reason, we will allow the job to be released five times but only allow it to fail—with a thrown exception—for two times. Here's how we'll do it:

```
class ProvisionServer implements ShouldQueue
{
    public $tries = 5;
    public $maxExceptions = 2;

    // ...
}
```

Using `$maxExceptions`, we configured the workers to fail the job if an exception was thrown from inside the `handle()` method on two different attempts. Now the job will be attempted five times; two of those five attempts could be due to an exception being thrown.

Handling Job Failure

We shouldn't forget about cleaning up when the job fails. So, inside the `failed()` method, we are going to create an alert for the user—to know that the server creation has failed—and delete the server from our records:

```
class ProvisionServer implements ShouldQueue
{
    // ...

    public function failed(Exception $e)
    {
        Alert::create([
            // ...
            'message' => "Provisioning failed!",
        ]);

        $this->server->delete();
    }
}
```

If we inspect the `failed_jobs` database table after a failed job, we'll see the job record along with the full stack trace of the exception. This will help us understand what happened.

If the job failed due to an issue with the Forge API, the exception message would show that in the `failed_jobs` records. However, if the job failed due to exhausting all allowed attempts, the exception record will have a message that says "Job has been attempted too many times."

In both cases, we'll need to investigate the issue and see why provisioning is taking too long or why the Forge API is throwing errors. This situation shows us the importance of monitoring the dead-letter queue.

Canceling a Conference

Unfortunately, conference organizers had to cancel every conference in 2020 due to a pandemic. But that gives us an exciting challenge to look into, which is refunding the tickets of all attendees of a very large conference.

Let's take a look at how a `CancelConference` job may look:

```
class CancelConference implements ShouldQueue
{
    public function __construct(
        private Collection $attendees
    ) {}

    public function handle()
    {
        $this->attendees
             ->each(function($attendee) {
                $attendee->invoice->refund();

                Mail::to($attendee)->send(...);
             });
    }
}
```

We loop over the attendees' list, refund the tickets using our billing provider's API, and finally send an email to notify each attendee.

Handling Timeouts

Since this is a huge conference, the `CancelConference` job may need several minutes to complete. However, Laravel has a timeout mechanism to prevent jobs from running indefinitely, as this may cause workers to get stuck and not process any job. This limit is set to 60 seconds by default. So, for our job, we need to instruct our workers to give the jobs more time:

```
class CancelConference implements ShouldQueue
{
    public $timeout = 18000;
}
```

Setting `$timeout = 18000` instructs Laravel to allow this job to run for 5 hours. Giving it enough time to refund all attendees. Under the hood, Laravel uses a PHP process control extension called PCNTL to install a timer inside the worker process. This timer starts counting once a job is picked up by the worker and sends a `SIGALRM` signal to the worker process. Laravel handles this signal and kills the worker process along with the job it's processing. We'll learn more about this in a future chapter.

Preventing Job Duplication

When a worker picks a job up from the queue, it marks it as `reserved` so no other worker picks that same job up. Then, once it finishes processing the job, it either removes it from the queue or releases it back to be retried. With that in mind, there's a risk that if the worker process crashes while in the middle of processing a job, this job will remain `reserved` forever and will never run.

To avoid this, Laravel sets a timeout for how long a job may remain in a `reserved` state. This timeout is 90 seconds by default, and it's set inside the `config/queue.php` configuration file on the connection level:

```
return [

    'connections' => [
        'database' => [
            // ...
            'retry_after' => 90,
        ],
    ];
];
```

Now our job may run for 5 hours, and Laravel will remove the `reserved` state in 90 seconds. That means another worker can pick this same job up while it's still being processed. This will lead to job duplication. A very bad outcome.

To avoid job duplication for long-running jobs, we have to always make sure the value of `retry_after` is **more** than any timeout we set on a worker level or a job level:

```
return [

    'connections' => [
        'database' => [
            // ...
            'retry_after' => 18060,
        ],
    ];
];
```

Setting `retry_after` to 5 hours + 1 minute here means the job will remain reserved for as long as a worker is processing it, eliminating the risk of job duplication.

Making Sure We Have Enough Workers

If that job takes 5 hours to finish, the worker processing it will be busy for 5 hours and unable to process any other jobs during that time. We mentioned earlier that a worker could only handle a single job at any time.

When we have long-running jobs in place, it's always a good idea to ensure we have enough workers to handle other jobs. However, even with having more workers, we could still get into a situation where all workers pick a 5-hour job up. And to avoid this, we need to push these long-running jobs to a special queue and assign just enough workers to it. All this while having other workers processing jobs from other queues.

If we dispatch our `CancelConference` job to a `cancelations` queue, assign five workers to that queue, and have another five workers processing jobs from the main queue, we'll only have 50% of our workers with a chance of being stuck at long-running jobs. So let's dispatch our `CancelConference` job to the `cancelations` queue:

```
CancelConference::dispatch($conference)
    ->onQueue('cancelations');
```

Then we start 5 workers to process jobs from that queue:

```
php artisan queue:work --queue=cancelations,default
                       --timeout=18000
```

> **Notice:** We configured the worker to process jobs from the `cancelations` *and* the `default` queues. If the `cancelations` queue is empty, the worker won't remain idle.

Using a Separate Connection

We had to set `retry_after` to 18060 a bit earlier. This will apply to all jobs dispatched to our `database` queue connection. So if a worker crashes while it's in the middle of processing **any** job, other workers will **not** pick it up for 5 hours.

For that reason, we need to set `retry_after` to 18060 seconds for just our `cancelations` queue. To do that, we'll need to create a completely separate connection in our

`config/queue.php` file:

```
return [
    'connections' => [
        'database' => [
            // 'driver' => 'database',
            // ...
            'retry_after' => 90,
        ],

        'database-cancelations' => [
            // 'driver' => 'database',
            // ...
            'retry_after' => 18060,
        ],
    ];
];
```

Now instead of dispatching our job to a `cancelations` queue, we're going to dispatch it to the `database-cancelations` connection:

```
CancelConference::dispatch($conference)
    ->onConnection('database-cancelations');
```

We can also configure the connection from within the job class:

```
class CancelConference implements ShouldQueue
{
    public $connection = 'database-cancelations';
}
```

Finally, here's how we will configure our workers to consume jobs from our new connection instead of the default one:

```
php artisan queue:work database-cancelations
```

> **Warning:** Workers cannot switch between connections. Those workers connected to the `database-cancelations` connection will only process jobs on that connection. If the connection has no jobs, the worker will remain idle.

Fault Tolerance Using Checkpointing

When designing queued jobs, we should always consider how we want the system to behave when things go wrong. We used Laravel's retry mechanisms in previous challenges to implement fault tolerance, which was simple and easy to reason about. However, when it comes to this `CancelConference`, retrying isn't as simple. We must consider how we want the system to behave when the job is retried. Is it going to re-refund customers that were already refunded? Is it going to send multiple emails? Is it going to loop over all conference attendees every time it retries?

To implement fault tolerance for our long-running job, we need to use a technique called *checkpointing*. It's saving a snapshot of the job's state so that it can be restarted from that point in case of failure.

To implement this technique, we need to keep track of the number of attendees we refunded and increment this number after each refund. We'll use the `Cache` facade for that:

```
use Illuminate\Support\Facades\Cache;

$this->attendees
     ->each(function($attendee) {
        $attendee->invoice->refund();

        Cache::increment('refunded');

        Mail::to($attendee)->send(...);
     });
```

And then we'll skip the refunded attendees from the loop:

```
use Illuminate\Support\Facades\Cache;

$this->attendees
     ->skip(Cache::get('refunded', 0))
     ->each(function($attendee) {
        // ...
```

```
        });
```

The `Cache::get()` call here will retrieve the value stored in the cache for the `refunded` key and fall back to `0` if there's no value stored there (No refunds occurred yet). We pass the result of this operation to the `skip()` collection method, which skips the first `n` records. In this case, `n` is the number of attendees already refunded. So now when this job retries, it will start from where it left, not from the beginning.

Finally, let's configure the job to attempt a few times:

```
class CancelConference implements ShouldQueue
{
    public $tries = 5;
}
```

Preventing a Double Refund

The `CancelConference` job from the last challenge may run for as long as 5 hours to refund all the attendees and send them an email. While doing so, the worker won't be able to pick up any other job since it can only process one job at a time. Moreover, if the worker crashes at any point, Laravel will wait for 18060 seconds (5 hours and 1 minute) to retry the job. Wouldn't it be better to dispatch multiple shorter jobs instead of a long-running one?

Let's see how we can do that:

```
$this->attendees
     ->each(function($attendee) {
         RefundAttendee::dispatch($attendee);
     });
```

Instead of dispatching a single `CancelConference` job from our controller, we will iterate over attendees and dispatch several `RefundAttendee` jobs.

Doing this has an additional—and significant—benefit, making it possible for multiple refund jobs to run concurrently, reducing the time needed to refund the attendees of a single conference.

Here's how the job will look:

```
class RefundAttendee implements ShouldQueue
{
    public $tries = 3;
    public $timeout = 60;

    public function __construct(
        private Attendee $attendee
    ) {}

    public function handle()
    {
        $this->attendee->invoice->refund();

        Mail::to($this->attendee)->send(...);
    }
}
```

Now this job does two things; refund the invoice and send an email. We have configured the job to allow 3 attempts in case of failure, which means we may retry the part where we refund the invoice multiple times. Ideally, our billing provider will take care of preventing a double refund, but why risk it?

It's always a good practice to avoid any negative side effects from the same job instance running multiple times. A fancy name for this is Idempotence.

Let's see how we can make our `RefundAttendee` job idempotent:

```
public function handle()
{
    if ($this->attendee->invoice->refunded){
        return;
    }

    $this->attendee->invoice->refund();

    $this->attendee->invoice->update([
        'refunded' => true
    ]);

    Mail::to($this->attendee)->send(...);
}
```

Before refunding the invoice, we'll check if it was already refunded. And after refunding,

we'll mark it as `refunded` in our database. That way, if the job was retried, it would not have the negative side effect of refunding the invoice multiple times.

Double Checking

When we call `refund` on an invoice, our code will send an HTTP request to the billing provider to issue a refund. If the refund was issued but a response wasn't sent back due to a networking issue, our request will error, and the invoice will never be marked as `refunded`.

The same will happen if a failure occurs between refunding the invoice and marking it as `refunded` in the database. When the job is retried, `refunded` will still be `false`.

In this situation, our billing provider is the source of truth. Only they know—for sure—if the invoice was refunded. So, instead of storing the `refunded` state in our database, it's more reliable to check if the invoice was refunded by contacting the billing provider:

```
public function handle()
{
    if ($this->attendee->invoice->wasRefunded()) {
        return;
    }

    $this->attendee->invoice->refund();

    Mail::to($this->attendee)->send(...);
}
```

In our example code, `wasRefunded()` and `refund()` hide the actual implementation of sending the HTTP requests to the billing provider.

Here's what the `wasRefunded()` method may look:

```
$response = HTTP::timeout(5)->get('../invoice/'.$id)
            ->throw()
            ->json();

return $response['invoice']['status'] == 'refunded';
```

When we call `wasRefunded()`, we check if that invoice was refunded on the billing provider's end, which is a more reliable source of information.

Triple Checking with Atomic Locks

You wouldn't willingly dispatch the `RefundAttendee` job multiple times for the same attendee, but you accidentally could. If two workers pick those 2 jobs up **at the same time**, you could end up sending multiple refund requests to the billing provider for the same attendee.

Figure 2-14 shows a logical view of such an outcome. When the two jobs checked if the attendee was already refunded, they both got a "no!" response since the billing provider hasn't actually refunded it yet. So, both jobs sent two concurrent `refund()` requests to the provider.

This pattern is called *check-then-act*, a very common source of race conditions that cause bugs known as *Time-of-check to time-of-use*. Or *TOCTOU* for short. In this pattern, a stale observation (refund status) is used to decide what to do next.

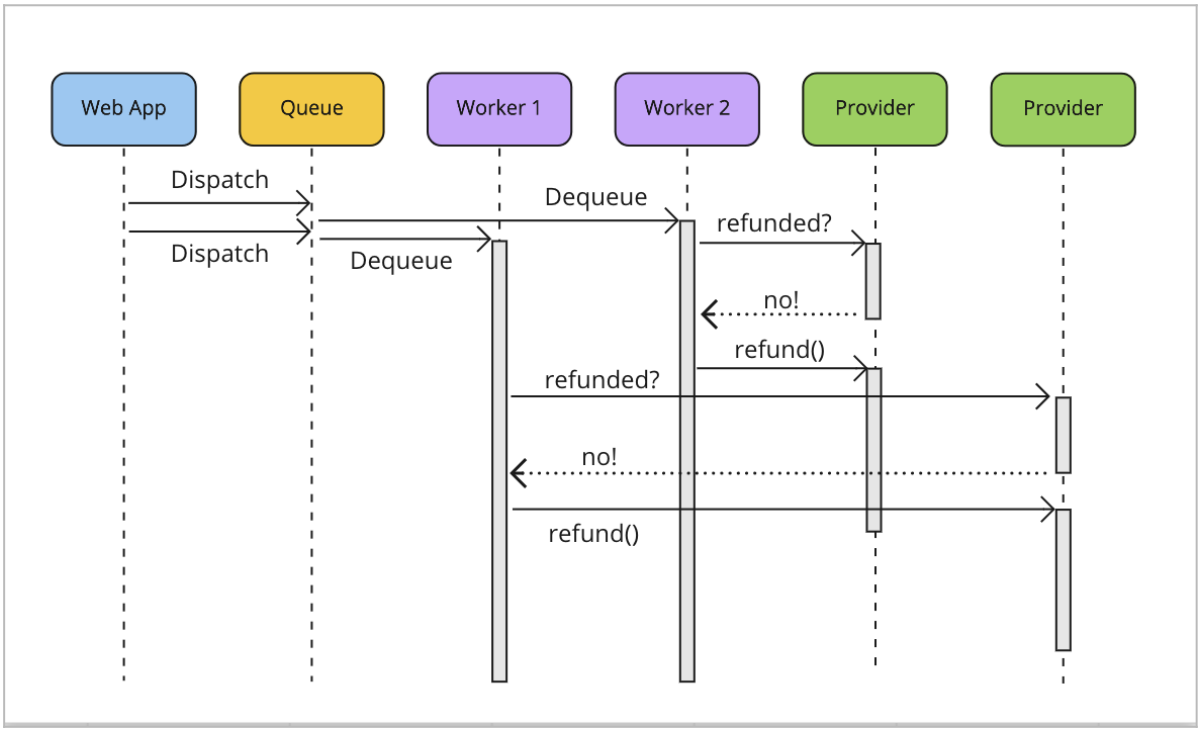

Figure 2-14. Two jobs were dispatched to refund the same attendee.

One way to avoid a TOCTOU bug is utilizing synchronization mechanisms such as atomic locks. Such a mechanism prevents critical sections of our code from being executed concurrently.

To use atomic locks in Laravel, we can rely on the `Cache` facade:

```
public function handle()
```

```
{
    $invoice = $this->attendee->invoice;

    Cache::lock('refund.'$invoice->id)
        ->get(function () use ($invoice) {
            if ($invoice->wasRefunded()) {
                return;
            }

            $invoice->refund();

            Mail::to($this->attendee)->send(...);
        });
}
```

In this example, before performing any work, we try to acquire a lock on the `refund.{id}` cache key. Only if a lock was acquired will we attempt to issue the refund. If another worker has already acquired this lock, the job will perform nothing and return.

> **Notice:** This lock will be automatically released after the mail is sent or if an exception was thrown.

Figure 2-15 shows what happens behind the scenes when we put an atomic lock in place. Since worker two was able to acquire the lock first, worker 1 was unable to acquire it, so the job was canceled.

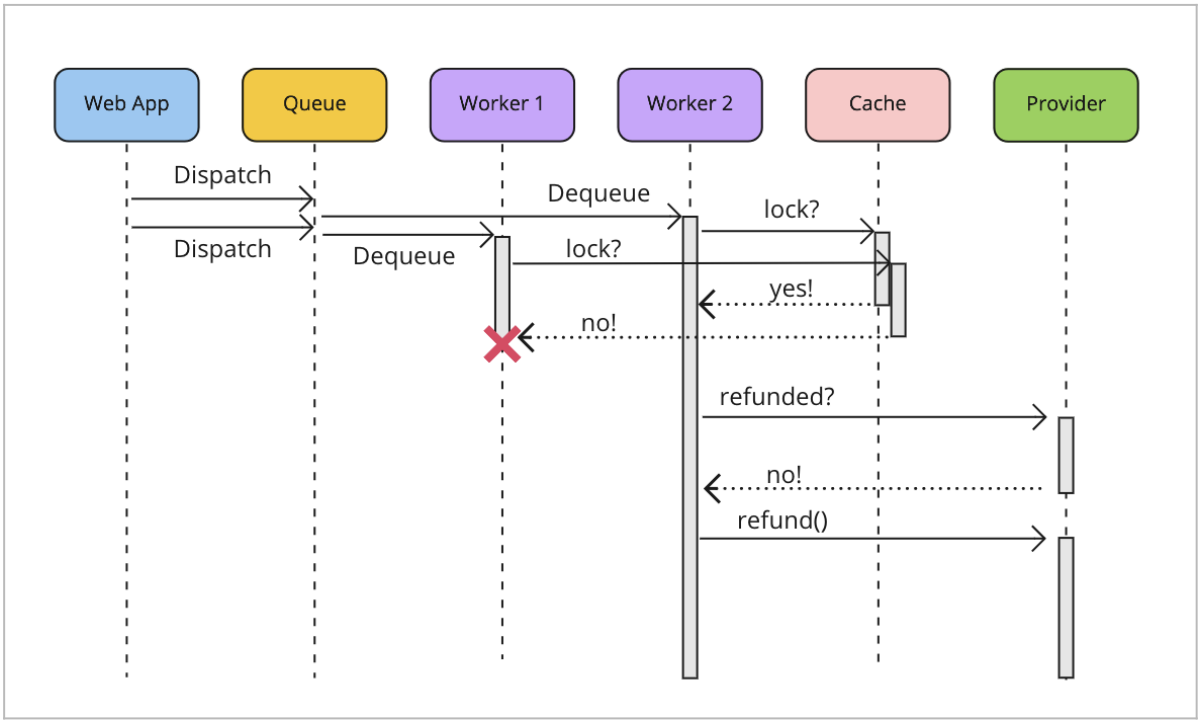

Figure 2-15. Atomic locks cancel the second job.

Managing Retries with Locks

Atomic locks are powerful tools that help us avoid bugs caused by race conditions. However, we should always consider a release mechanism for our locks to avoid stale locks laying around and preventing work from being done.

For example, imagine if the worker crashes somewhere after acquiring the lock in the `RefundAttendee` job and before doing the actual refunding. Since a lock is in place, future attempts of the job won't be able to acquire one, and thus the job will be marked as successful even though the refund didn't happen.

To avoid this, we need to set automatic expiration for the lock and ensure future retries of the job happen after the lock expires:

```
public function handle()
{
    $invoice = $this->attendee->invoice;

    Cache::lock('refund.'$invoice->id, 10)
            ->get(
                ...
            );
}
```

By providing a second argument to the `Cache::lock()` method, we are telling Laravel to expire the lock after 10 seconds automatically.

Now, if the worker crashes, the job will remain reserved for 90 seconds. By the time the job is retried, the lock will be expired, and we'll be able to acquire a new one to run the job.

> **Notice:** The job will be retried after 90 seconds because the default `retry_after` in our `config/queue.php` file is set to `90`.

Configuring Backoff

If an exception was thrown, the lock would be released automatically. But what if releasing the lock failed? In our situation, the lock will automatically expire after 10 seconds. During those 10 seconds, though, the job may be retried and finds the lock still in place and thus never completes the task. This is because when a job fails, Laravel will attempt to retry it immediately unless there's a backoff in place.

To avoid this, we're going to configure a proper backoff on the job:

```
public $backoff = 11;
```

Now, if the job fails, it'll be retried after 11 seconds. With the lock expiring after only 10 seconds, the retry will find the lock available and will be able to perform the task.

> **Notice:** If the job fails, it'll be retried after 11 seconds. But if the worker crashes, the job will be retried after 90 seconds.

Preventing a Double Refund With Unique Jobs

In the previous challenge, we used atomic locks to prevent running a `RefundAttendee` job instance multiple times for the same invoice.

In Laravel 8.14.0 (Released on November 10, 2020), a new `ShouldBeUnique` job interface

was introduced. When a job implements this interface, Laravel will only allow one instance of this job to be dispatched to the queue:

```
use Illuminate\Contracts\Queue\ShouldBeUnique;

class RefundAttendee implements ShouldBeUnique
{
    public function __construct(
        private Attendee $attendee
    ){}

    public function handle()
    {
        if ($this->attendee->invoice->wasRefunded()) {
            return;
        }

        $this->attendee->invoice->refund();

        Mail::to($this->attendee)->send(...);
    }
}
```

If we try to dispatch this job multiple times, Laravel will only dispatch it once. It will ignore all other dispatches as long as an instance of the job is still in the queue. Under the hood, Laravel tries to acquire an atomic lock while dispatching the job (Figure 2-16). It uses the job class name as the lock key and keeps the lock in place until the job either succeeds or fails.

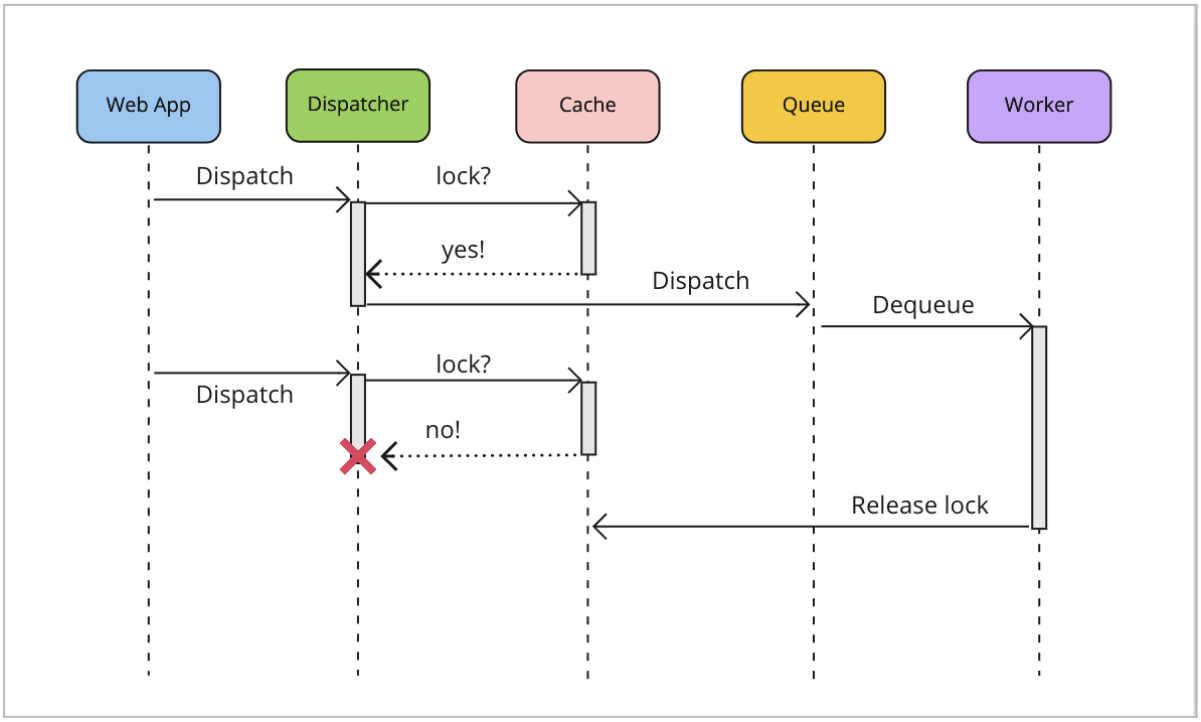

Figure 2-16. Preventing multiple dispatches of the same job.

However, our goal is not to prevent duplicate instances of `RefundAttendee` from being in the queue. The goal is to prevent duplicate instances of `RefundAttendee` that refund the **same** attendee. To configure this, we're going to implement a `uniqueId()` method:

```
use Illuminate\Contracts\Queue\ShouldBeUnique;

class RefundAttendee implements ShouldBeUnique
{
    public function uniqueId()
    {
        return $this->attendee->invoice->id;
    }
}
```

Now, if we accidentally dispatch `new RefundAttendee(Attendee::find(100))` multiple times, Laravel will only consider the first dispatch and ignore the others.

Configuring Lock Expiration

As we shared earlier, we should *always* consider a release mechanism for our locks to avoid stale lock situations. In our case, a stale lock may happen if the dispatching failed after the lock was acquired. In that case, the lock will remain in place, and nothing will release it.

To avoid that, we should automatically expire the lock after some time by setting the `$uniqueFor` public property on the job class:

```
use Illuminate\Contracts\Queue\ShouldBeUnique;

class RefundAttendee implements ShouldBeUnique
{
    public $uniqueFor = 5 * 60;
}
```

Now the lock will auto-expire 5 minutes after the job is dispatched. And with that in mind, it's essential to consider the time you'd expect the job to be in the queue. If the job remains longer than the `$uniqueFor` value, the lock will expire, and other instances of the same job for the same attendee will make it to the queue.

Figure 2-17 illustrates this scenario. The job was dispatched at 00:00 with `uniqueFor` set to expire after 5 minutes. At 02:00, the job was processed and released back to the queue for retrying. At 05:00, the lock expired while the job was still retrying. Finally, at 07:00, another duplicate was dispatched, and both the former and the new job ran at 09:00.

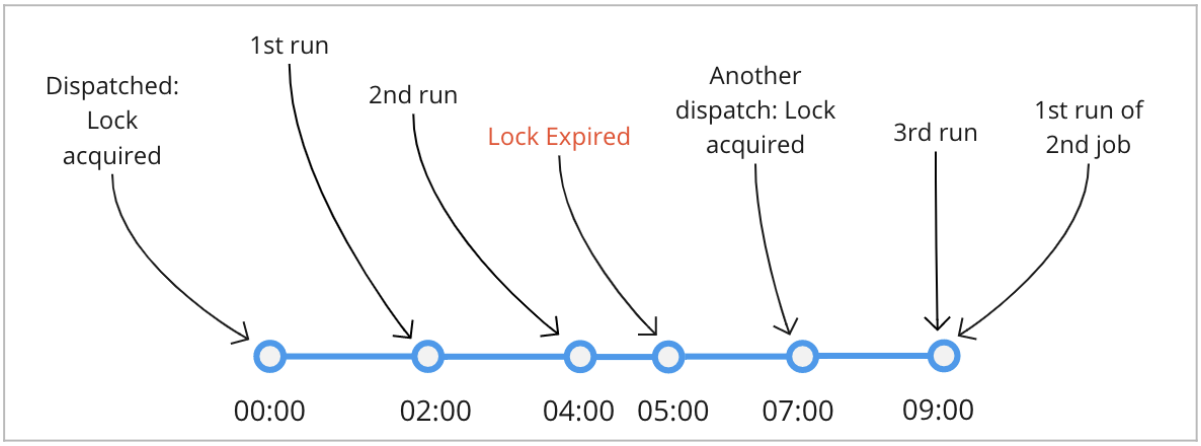

Figure 2-17. The lock expired before the job finished.

Since there's no guarantee a job will be processed during a given time, I don't recommend using unique jobs for this kind of critical section. Instead, use atomic locks inside the job as we did in the last challenge. With atomic locks, the lock is acquired right before processing the job, so it guarantees no other job acquires the lock even if multiple jobs are in the queue already.

Using the `ShouldBeUnique` interface can be helpful for jobs in which uniqueness is considered as best effort rather than a guarantee. These are jobs in which no damage will happen if duplicates exist. An example would be a job that generates a report. If the report

generation ran multiple times, there would be no damage. Trying to achieve uniqueness here is just a way to reduce the waste of consuming the system resources numerous times.

Figures 2-18 and 2-19 represent the difference between the two approaches. When using atomic locks, the lock is acquired right before entering the critical section, refunding the attendee. And thus, there's way less chance of the lock expiring while the job is still processing.

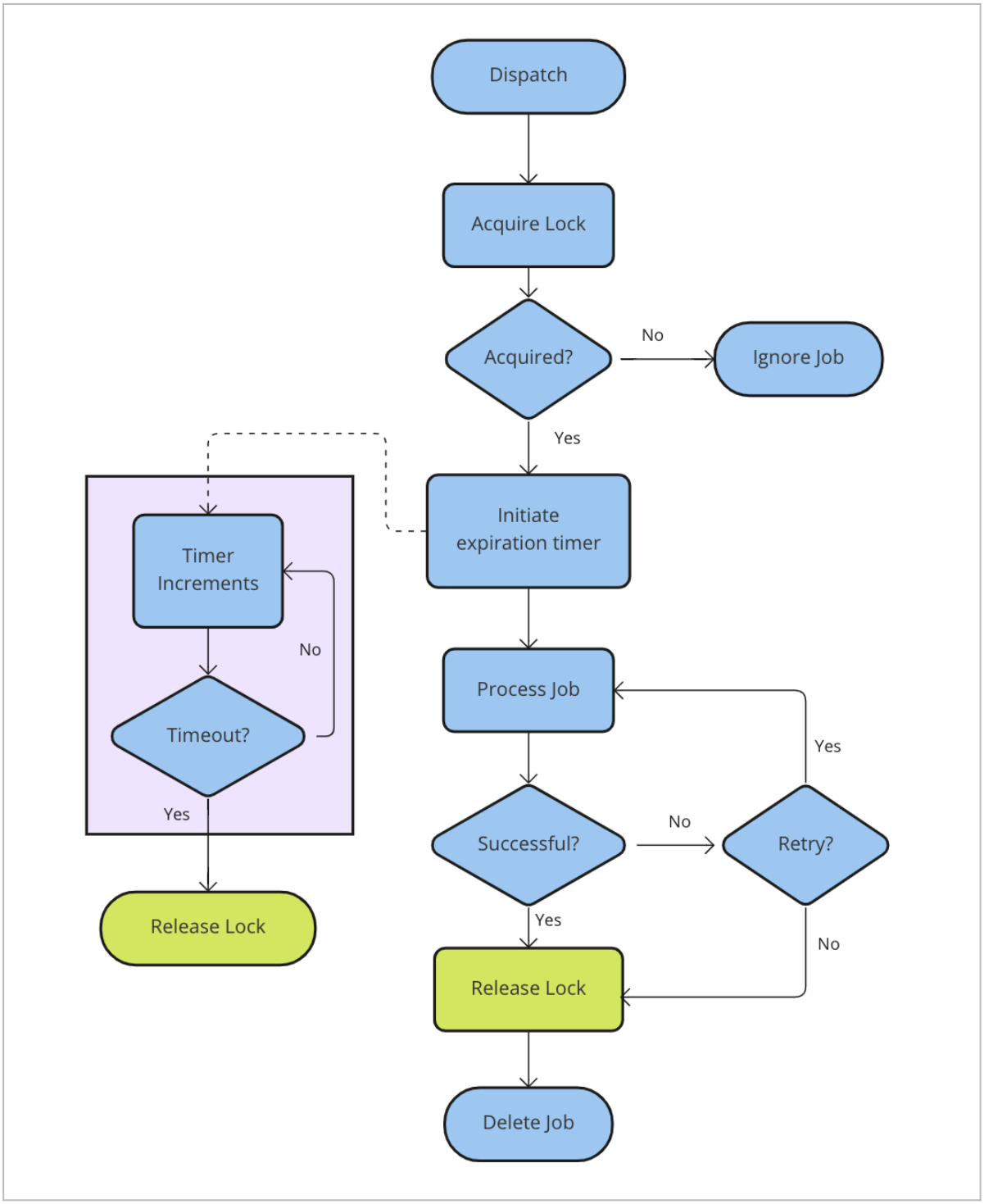

Figure 2-18. Using unique jobs.

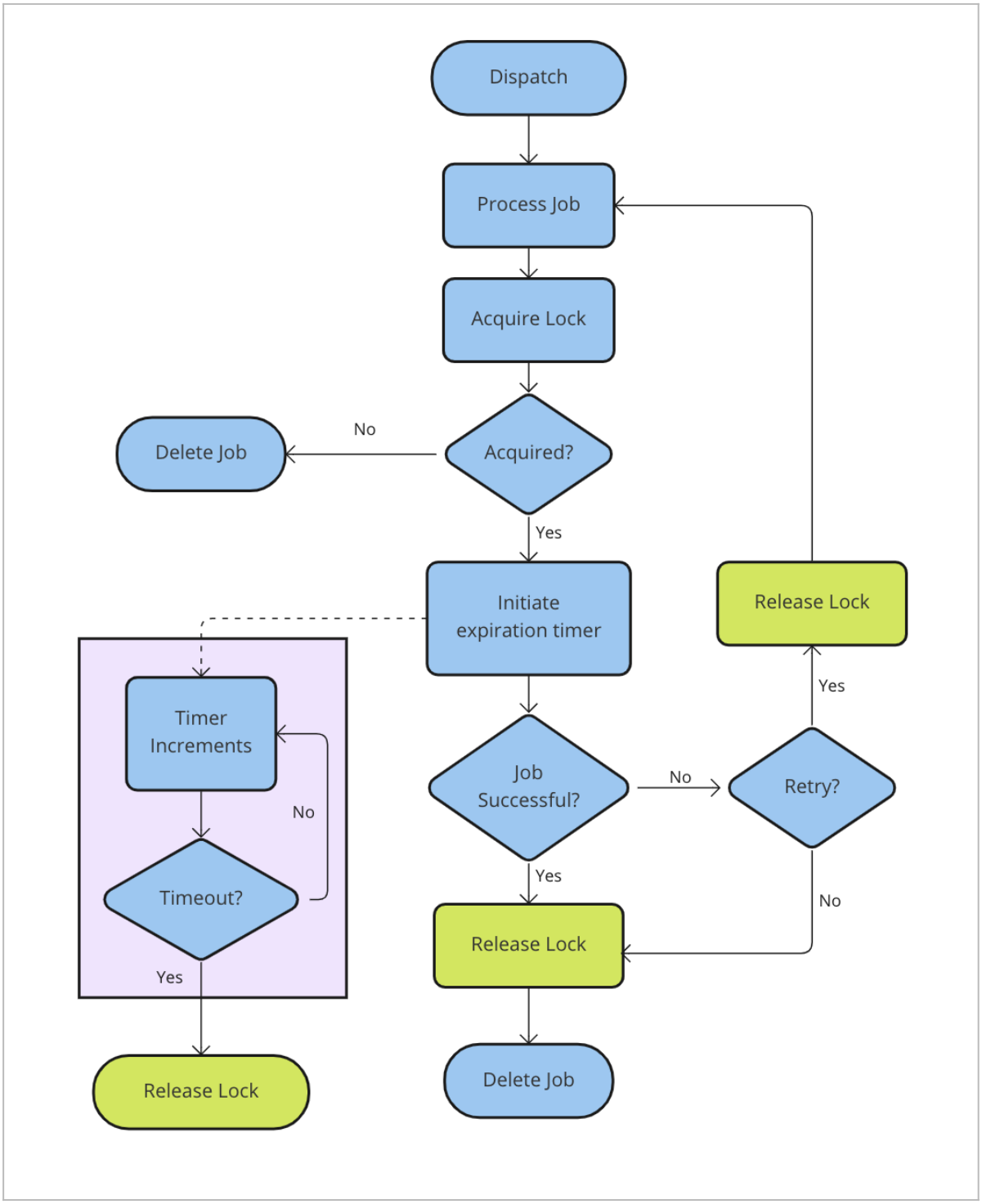

Figure 2-19. Using atomic locks while processing the job.

Bulk Refunding Invoices

In a previous challenge, we learned why it's a good idea to dispatch multiple shorter jobs instead of a long-running one. However, this approach has one primary concern, especially when the number of jobs is significant. For each dispatch operation, an instruction is sent to

our queue store. This means we're performing multiple round-trips between our web application and the store, which is not an efficient way to do it.

To avoid this, we may use the `bulk()` method of the `Queue` facade and provide an array of jobs. This method will prepare the records of all jobs and send them to the queue store in one go:

```
use Illuminate\Support\Facades\Queue;
use App\Jobs\RefundAttendee;

$jobs = $this->attendees
             ->map(function($attendee) {
                 return new RefundAttendee($attendee);
             })
             ->toArray();

Queue::bulk($jobs);
```

> **Notice:** The `bulk()` method sends multiple jobs in one command only when using the database or Redis drivers.

Using Job Batching

Another way of bulk dispatching in Laravel is by using job batches. This feature allows us to efficiently dispatch multiple jobs and monitor and control those jobs while in the queue.

To use job batches with our `RefundAttendee` job, we need to add the `Batchable` trait to it:

```
// ...
use Illuminate\Bus\Batchable;

class RefundAttendee implements ShouldQueue
{
    use Batchable;
    // ...
}
```

Next, Laravel stores the batch information in a database table. And this table doesn't exist by

default. So, before we can use batches, we need to run the following command:

```
php artisan queue:batches-table
```

This will create the migration for the `job_batches` database table. Now let's run the migration:

```
php artisan migrate
```

Inside this new table, a record may look like this:

```
{
    "id": "67b31b39-26df-4b39-b2db-c4fb78088fd1",
    "name": "",
    "total_jobs": 400,
    "pending_jobs": 400,
    "failed_jobs": 0,
    "failed_job_ids": [],
    "options": "",
    "created_at": 1591425946,
    "cancelled_at": null,
    "finished_at": null,
}
```

Laravel stores the batch's UUID, a descriptive name, the total number of jobs, the number of jobs still in queue (pending), the number of failed jobs and their IDs, the batch options (more on that later), and the timestamps of the batch creation, cancelation, and completion.

Now to dispatch a batch, you may use the `batch()` method of the `Bus` facade:

```
use Illuminate\Support\Facades\Bus;

$jobs = /** ... **/;

Bus::batch($jobs)->dispatch();
```

And you can set a descriptive name for the batch using the `name` method:

```
Bus::batch($jobs)
```

```
    ->name('Cancel Conference '. $conference->id)
    ->dispatch();
```

The `dispatch()` method returns an instance of `Illuminate\Bus\Batch`. We can extract the batch ID from this instance and store it in the conference model for future reference:

```
$batch = Bus::batch($jobs)
            ->name('Cancel Conference '. $conference->name)
            ->dispatch();

$conference->update([
    'refunds_batch' => $batch->id
]);
```

Monitoring the Refunding Process

In many situations, you can assume that dispatching jobs is a fire-and-forget action. In our refunding challenge, however, we must monitor the progress of the refund process. And even though each of the `RefundAttendee` jobs is processed separately, Laravel sits there and watches everything.

Using the reference to the batch we stored earlier, we can show the conference organizer information about the progress of the refund process. To retrieve the batch object using its ID, we may use the `findBatch()` method of the `Bus` facade:

```
$batch = Bus::findBatch(
    $conference->refunds_batch
);

return [
    'progress' => $batch->progress().'%',
    'remaining_refunds' => $batch->pendingJobs,
    'has_failures' => $batch->hasFailures(),
    'is_cancelled' => $batch->canceled()
];
```

Canceling a Batch

Using the reference to the `Batch` object, we can also allow the user to cancel the refunding process at any time. To do that, we may call the `cancel()` method:

```
$batch = Bus::findBatch(
    $conference->refunds_batch
);

$batch->cancel();
```

Calling `cancel()` on the batch will mark it as `canceled` in the database, but it will not prevent any jobs still in the queue from running. Therefore, we need to check if the batch is still active before running any job code. So, inside the `handle()` method of our `RefundAttendee` job, we'll add that check:

```
public function handle()
{
    if ($this->batch()->canceled()) {
        return;
    }

    // Actual refunding code here ...

    $this->attendee->update([
        'refunded' => true
    ]);
}
```

When a worker processes a job that belongs to a canceled batch, the job will return, and the worker will consider it successful and remove it from the queue.

> **Notice:** By setting a `refunded` field to `true` in the attendee model, we will be able to know which attendees were refunded before the batch was canceled.

Handling Batch Jobs Failure

With the default configurations, a batch will get canceled automatically if any jobs fail—with no more retries—. That means if we refund 1000 attendees and the job for attendee #300 fails, the rest of the jobs will find the batch marked as canceled and will not perform the refund. This can be useful in some cases but not in this one.

To force the batch to continue even if some of the jobs fail, we may use the `allowFailures()` method:

```
$batch = Bus::batch($jobs)
        ->allowFailures()
        ->dispatch();
```

Now Laravel will keep the batch going without canceling it in case of failure. In addition, it will also store the IDs of the failed jobs in the `job_batches` database table.

You can retry any of the failed jobs using the `queue:retry` command:

```
php artisan queue:retry {job_id}
```

And you can retry all failed jobs of a given batch using the `queue:retry-batch` command:

```
php artisan queue:retry-batch {batch_id}
```

Notifying the Organizer

Since the refunding process can take a while, it would be nice if we notified the conference organizer when important events happen during the refunding process. The `Batch` object exposes a `then()`, `catch()` and `finally()` methods that we can use to send the notifications:

Let's start with a notification to be sent after all attendees were refunded. To send that notification, we may use the `then()` method:

```
$organizerEmail = $conference->organizer->email;

Bus::batch($jobs)
    ->allowFailures()
    ->then(function () use ($organizerEmail){
        Mail::to($organizerEmail)->send(
            'All attendees were refunded successfully!'
        );
    })
    ->dispatch();
```

First, we store the conference organizer's email address in a `$organizerEmail` variable. Then call the `then()` method and provide a closure. Inside it, we use the `Mail` facade to

send the mail to the email address stored inside the `$organizerEmail` variable.

The reason we stored the email address in a variable is that Laravel serializes the closures provided to the `then()`, `catch()` and `finally()` methods and stores this data inside the `options` column of the `job_batches` database. It is a best practice to keep the closure scope minimal, so the serialization process is more efficient. A variable carrying a string is easier to serialize than a variable carrying a `Conference` model instance.

Now, If all the jobs in the batch finish successfully, Laravel will execute any callback provided to the `then()` method. In this case, our callback will send an email to the organizer.

We can also notify the organizer when a failure happens in one of the batch jobs. To do that, we may use the `catch()` method and provide a closure:

```
Bus::batch($jobs)
    ->allowFailures()
    ->then(...)
    ->catch(function () use ($organizerEmail){
        Mail::to($organizerEmail)->send(
            'We failed to refund some of the attendees!'
        );
    })
    ->dispatch();
```

Since we use `allowFailures()` here, the batch jobs will continue processing normally even if a single job fails. But it's a nice experience to notify the organizer right away that a refund has failed.

Finally, after all the batch jobs are done, the `finally()` method will be called. Some of these jobs might have been completed successfully, while others might have failed. We can use this method to notify the organizer that the process is now complete:

```
Bus::batch($jobs)
    ->allowFailures()
    ->then(...)
    ->catch(...)
    ->finally(function () use ($organizerEmail){
        Mail::to($organizerEmail)->send(
            'Refunding attendees completed!'
        );
    })
    ->dispatch();
```

Dispatching a Large Batch

When you dispatch many jobs in bulk, Laravel prepares the payload in memory and sends it over the network. Your queue store will receive this large payload and start persisting it. This can take a lot of time and affect the user experience since users will have to wait for the batch to be dispatched before the web app can respond to them.

Figure 2-20 illustrates the problem. The web server will wait for the queue store to persist the batch before sending a response back to the user.

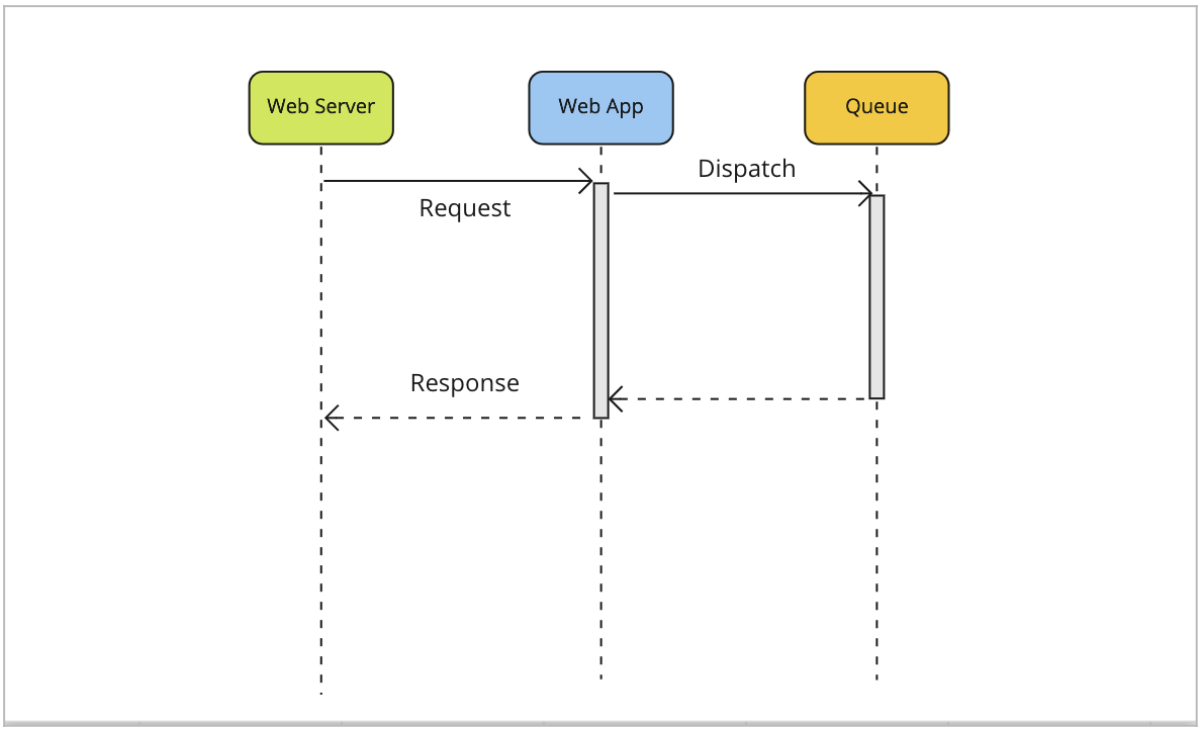

Figure 2-20. Dispatching a large batch blocks the execution.

To solve the issue, we can use `dispatchAfterResponse()` instead of `dispatch()` to dispatch a batch. When using this method, Laravel will delay the dispatching process until after sending the response to the user:

```
Bus::batch($jobs)
    // ...
    ->dispatchAfterResponse();
```

Even though this makes your application feel faster for the end user, the PHP process that handles the request will remain blocked and won't be able to handle other requests until the dispatching finishes (Figure 2-21). This might reduce the throughput of your web server if

enough processes got blocked by dispatches of large batches.

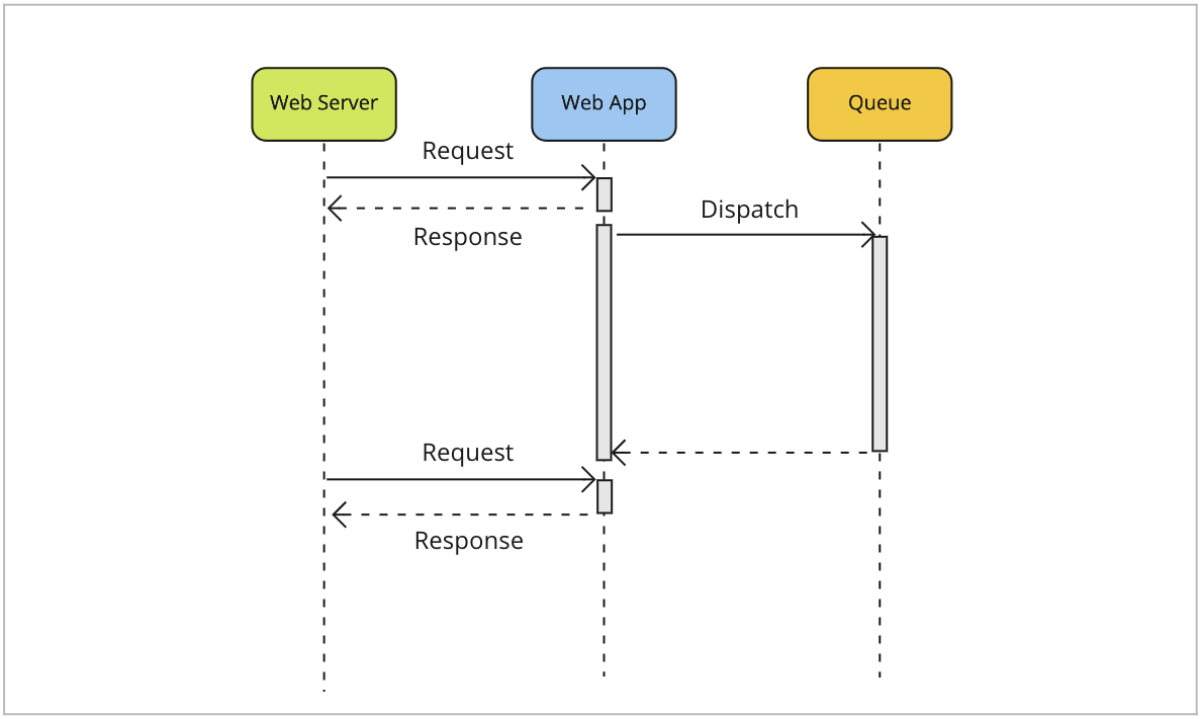

Figure 2-21. The web app process is blocked while dispatching a large batch.

Now let's take a look at another approach, which is dispatching the batch in chunks.

```
$jobs = $this->attendees
             ->take(100)
             ->map(function($attendee) {
                 return new RefundAttendee($attendee);
             })
             ->toArray();

$loaders = $this->attendees
            ->skip(100)
            ->chunk(100)
            ->map(function ($attendees) {
                return new ContinueRefundingAttendees($attendees);
            })
            ->toArray();

Bus::batch(array_merge($jobs, $loaders))->dispatch();
```

First, we'll convert only the first 100 attendees from the collection to `RefundAttendee` jobs. Then, We'll transform the remaining attendees into chunks of 100 and each chunk to a `ContinueRefundingAttendees` job. Then while dispatching the batch, we merge the two

arrays of jobs in one batch. So now the batch will contain 100 `RefundAttendee` jobs and several `ContinueRefundingAttendees` jobs, each with reference to 100 attendees.

Inside the `handle()` method of the `ContinueRefundingAttendees` job, we will add jobs for the 100 attendees to the batch using the `add()` method of the `Batch` object:

```
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $jobs = $this->attendees
              ->map(function($attendee) {
                  return new RefundAttendee($attendee);
              })
              ->toArray();

    $this->batch()->add($jobs);
}
```

> **Warning:** Warning: Remember to add the `Batchable` trait to the `ContinueRefundingAttendees` so you can access the Batch object.

With this approach, we're dispatching a small number of jobs to the batch. Some of these jobs will add more jobs to the batch to refund the remaining attendees. That way, the web app won't take much time to dispatch the batch.

Minding Timeouts

For job batching to work, Laravel needs to do some housekeeping after processing any job (Figure 2-22), whether the job is completed or failed. This work is done right before deleting the job from the queue and moving to process another job.

- If the current job is the last in the batch, `finally` callbacks are called.
- If the current job is the first failed job, `catch` callbacks are called.
- If the current job is the last and there aren't any failed jobs, `then` callbacks are called.

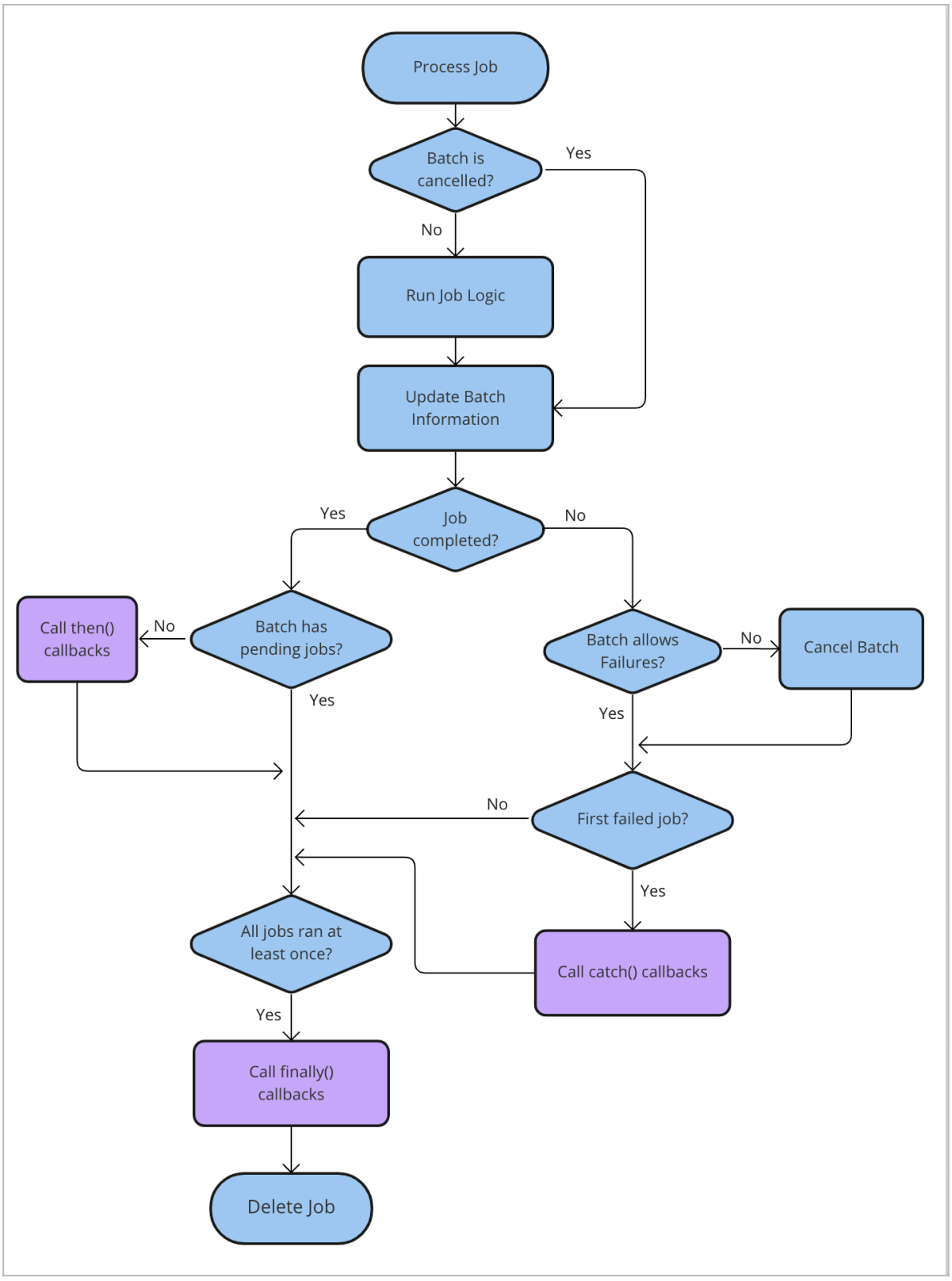

Figure 2-22. The housekeeping after every batch job.

If you have a timeout set for a job that will call any batch callbacks, the time spent making these calls will be counted on the job. In other words, the job may timeout while in the middle of calling these callbacks.

When this happens, the job may be retried while it has already completed refunding the attendee. The worker will think the job failed while it finished. It's just that it timed out while Laravel was doing the housekeeping for the batch.

So, when dispatching jobs to a batch. Set a proper timeout for all jobs, so they may handle the housekeeping without timing out. Also, ensure not to include any time-consuming tasks within the batch callbacks. For example, instead of sending the notification mail inside the callback, dispatch a job instead:

```
Bus::batch($jobs)
    ->finally(function () use ($organizerEmail){
        SendCompletedNotification::dispatch($organizerEmail);
    })
    ->dispatch();
```

By doing so, there's less chance the job will timeout due to batch housekeeping taking too long.

Selling Conference Tickets

The pandemic is over! Everyone is looking forward to the joy of attending conferences and meeting interesting people.

Now we need to write the controller action for selling conference tickets. Let's see how that may look:

```
public function store(Conference $conference)
{
    $attendee = Attendee::create([
        'conference_id' => $conference->id
    ]);

    $providerInvoice = BillingProvider::charge([
        'reference' => $invoice->id,
        // ...
    ]);

    $attendee->update([
        'provider_invoice_id' => $invoice->id
    ]);
}
```

1. First, we create an `Attendee` model instance using the user information.
2. Then, we use our billing provider SDK to create an invoice and charge the attendee for the ticket.
3. Finally, we update the attendee with reference to the provider invoice.

Inside the `Attendee` model, we may watch the `provider_invoice_id` for changes and dispatch a `SendTicketInformation` job. This job sends an email to the user with the ticket information:

```
protected static function boot()
{
    parent::boot();

    Attendee::updated(function ($attendee) {
        if ($attendee->wasChanged('provider_invoice_id') &&
            $attendee->provider_invoice_id){
                SendTicketInformation::dispatch($attendee);
        }
    });
}
```

Here we're listening to the `updated` model event inside the `boot` method of the model class and checking if the `provider_invoice_id` was populated. If that's the case, we dispatch the `SendTicketInformation` event.

Let's consider the case when our billing provider fails to charge the user. Usually, an exception will be thrown out of the `BillingProvider::invoice()` method, and the user will get an error.

Because an exception was thrown, the attendee's `provider_invoice_id` attribute will not get populated, and the `SendTicketInformation` job will not get dispatched. However, the attendee model was already created, and the user is now registered as an attendee! That's not right.

To solve this issue, you may consider using a database transaction to rollback any database changes if an exception was thrown:

```
public function store(Conference $conference)
{
    DB::transaction(function() use ($conference) {
```

```
        $attendee = Attendee::create([
            'conference_id' => $conference->id
        ]);

        $providerInvoice = BillingProvider::charge([
            'reference' => $invoice->id,
            // ...
        ]);

        $attendee->update([
            'provider_invoice_id' => $invoice->id
        ]);
    });
}
```

Now, if an exception is thrown anywhere inside the transaction callback, the whole transaction will be reverted, and the attendee record will not exist.

Workers Are Very Fast

This section of our program starts a database transaction, updates some records, charges the user, dispatches a queued job, and then commits the transaction (Figure 2-24). It seems like a straightforward sequential flow of data and commands until you notice the worker on the far right of the diagram.

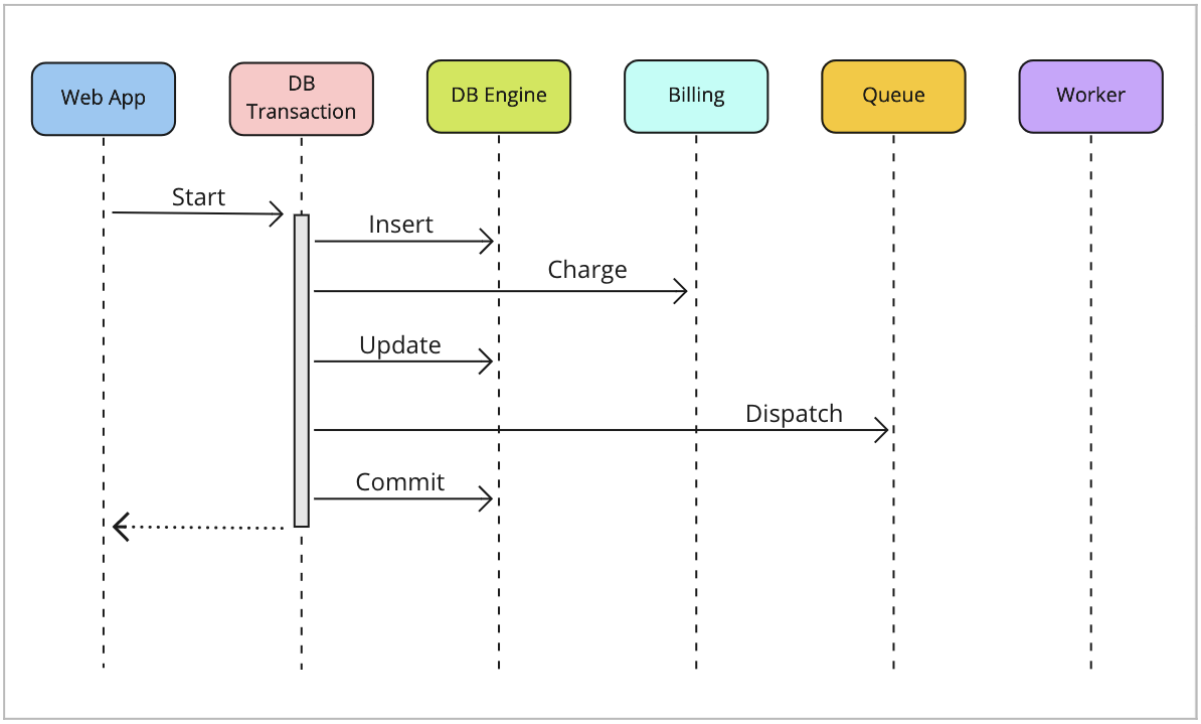

Figure 2-23. The process of selling a ticket.

We know from earlier that workers run in separate PHP processes where they keep dequeuing and processing jobs. That means a worker may pick up the `SendTicketInformation` job and start processing it before the database transaction commits (Figure 2-25). When the job attempts to retrieve the attendee record from the database, a `ModelNotFoundException` exception will be thrown, and the job will be moved to the dead-letter queue.

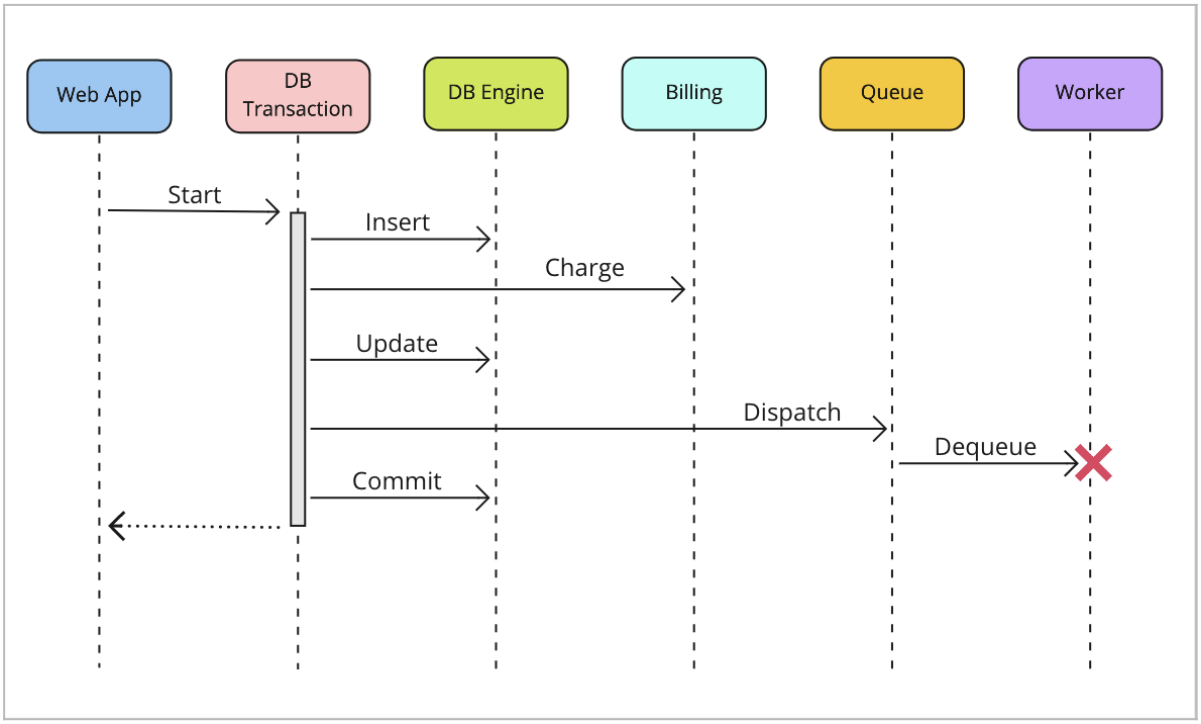

Figure 2-24. The worker picks the job up before the transaction commits.

The attendee record won't exist in the database engine until the transaction commits. So if the job starts too soon, the engine won't find it.

You may think configuring a job retry policy with a proper backoff will ensure the job retries after the transaction commits and eventually completes successfully. But that's not what will happen.

When Laravel detects a `ModelNotFoundException`, it moves the job straight to the dead-letter queue and disables any retry policies. Because, to Laravel, if a model doesn't exist now, it won't exist if it retries. Laravel doesn't account for this race condition situation. If you manually retry this job using the `queue:retry` command, it will complete successfully.

> **Notice:** If you're using the database queue driver, the dispatching part will be within the database transaction, and you won't face that issue. The job will only be added to the "jobs" database table **after** the transaction has been committed.

To avoid this race condition, we can dispatch the job with a delay that's long enough to guarantee the transaction has been committed:

```
protected static function boot()
{
    parent::boot();
```

```
    Attendee::updated(function ($attendee) {
        if ($attendee->wasChanged('provider_invoice_id') &&
            $attendee->provider_invoice_id){
                SendTicketInformation::dispatch($attendee)->delay(5);
        }
    });
}
```

Adding `delay(5)` here will make sure the job doesn't get picked up by workers for 5 seconds. This gives the transaction enough time to commit and the attendee record to persist in the database.

However, this also means the job will get dispatched even if the transaction was rolled back. In that case, Laravel will catch the `ModelNotFoundException` and move the job to the dead-letter queue. This is not too bad, but we're polluting the queue with a job that will fail and the dead-letter queue with a job we may not retry.

We can tell Laravel to delete the job if a `ModelNotFoundException` was thrown instead of moving it to the queue. Setting the `$deleteWhenMissingModels` property on the job class here instructs Laravel to do that:

```
class SendTicketInformation implements ShouldQueue
{
    public $deleteWhenMissingModels = true;
}
```

Now we're left with just the negative side effect of polluting the queue with a job that will fail.

Dispatching After Transactions Commit

Luckily, Laravel 8.19.0 (Released on December 15, 2020) came with a solution to this problem:

```
protected static function boot()
{
    parent::boot();

    Attendee::updated(function ($attendee) {
        if ($attendee->wasChanged('provider_invoice_id') &&
            $attendee->provider_invoice_id){
```

```
                SendTicketInformation::dispatch($attendee)->afterCommit();
            }
        });
    }
```

Using the `afterCommit()` method while dispatching the job instructs Laravel not to dispatch it until any open transaction commits. If the transaction was rolled back, the job would not get dispatched, which is exactly what we want.

We can also configure Laravel to delay dispatching a certain job until transactions commit by setting a `$afterCommit` public property on the job class:

```
class SendTicketInformation
{
    public $afterCommit = true;
}
```

Or set the `after_commit` connection option in your queue connection's configuration array:

```
'redis' => [
    'driver' => 'redis',
    // ...
    'after_commit' => true,
],
```

Now Laravel will delay dispatching any jobs on this connection until all transactions commit.

Spike Detection

Now let's move to a new challenge where we'll learn about a fundamental characteristic of asynchronously executed tasks. Unfortunately, people usually forget about this one, which can cause a lot of confusion.

In this challenge, we're looking at a monitoring platform customers use to get live updates on their websites. And while having a spike in traffic is great news for a customer, we must alert them about it, so they ensure they have enough resources to handle it. Imagine spending months trying to gain traction for your site, and suddenly someone popular talks about it. Then, it receives a sudden traffic spike that it can't handle, and requests start timing

out. Not good.

The code we may write to send such alerts will look like this:

```
foreach (Site::all() as $site)
{
    if ($site->current_visitors >= $site->threshold) {
        SendSpikeDetectionNotification::dispatch($site);
    }
}
```

We iterate over all the existing sites and dispatch a job if the number of current site visitors crosses a certain threshold. This code could be running within a CRON task that executes every minute.

The `SendSpikeDetectionNotification` job expects an instance of a `Site` model as a constructor argument. Here's how the job class may look:

```
class SendSpikeDetectionNotification implements ShouldQueue
{
    public function __construct(
        private Site $site
    ) {}

    public function handle()
    {
        SMS::send(
            $this->site->owner,
            "Spike detected on {$this->site->name}!
            Current visitors: {$this->site->current_visitors}"
        );
    }
}
```

When this job runs, the site owner will receive an SMS message with the site name and number of current visitors.

While this logic is valid, there is no guarantee that workers will pick this job up immediately. If the queue is busy, the job may get picked up after a while. By that time, the site's value of `current_visitors` may have changed.

If the user has the threshold set to 100,000, for example, our job may send an SMS that says:

> Spike detected on yoursite.com! Current visitors: 70,000

This is quite confusing, and our users may think the alerting system is buggy and notifying too early.

Model Serialization

As we explained earlier, Laravel converts each job to string form so it can be stored in the queue. When our `SendSpikeDetectionNotification` job is dispatched, the payload in the queue storage will look like this:

```
{
   "uuid":"765434b3-8251-469f-8d9b-199c89407346",
   // ...
   "data":{
      "commandName":"App\\Jobs\\SendSpikeDetectionNotification",
"command":"O:16:\"App\\Jobs\\SendSpikeDetectionNotification\":9:{s:22:\"\u0000App\\Jobs\
\SendSpikeDetectionNotification\u0000site\";O:45:\"Illuminate\\Contracts\\Database\\Mode
lIdentifier\":4:{s:5:\"class\";s:8:\"App\\Site\";s:2:\"id\";i:1;s:9:\"relations\";a:0:{}
s:10:\"connection\";s:5:\"mysql\";}s:3:\"job\";N;s:10:\"connection\";N;s:5:\"queue\";N;s
:15:\"chainConnection\";N;s:10:\"chainQueue\";N;s:5:\"delay\";N;s:10:\"middleware\";a:0:
{}s:7:\"chained\";a:0:{}}"
   }
}
```

The `data.command` attribute holds a serialized version of the `SendSpikeDetectionNotification` instance. I know it's hard to locate, but in the serialized version, the `site` class property holds an instance of `ModelIdentifier`, not `App\Site`.

A `ModelIdentifier` is a simple instance that holds the model ID, the class name, the name of the database connection, and the names of any relationships loaded. The worker uses this information to retrieve the model from the database when it picks the job up:

```
class ModelIdentifier
{
    public function __construct(
        $class,
        $id,
        $relations,
```

```
            $connection
        )
        {
            $this->id = $id;
            $this->class = $class;
            $this->relations = $relations;
            $this->connection = $connection;
        }
    }
```

Laravel does that for several reasons. Most importantly, to reduce the size of the job payload and prevent issues that may arise from attempting to serialize and un-serialize the model object. An eloquent model is a heavy object, and we want to avoid serializing it.

This magic is done inside the `SerializesModels` trait that's added by default to every Job class generated in your Laravel application. It's highly recommended that you keep that trait if your job accepts an Eloquent Model in the constructor.

Making the Job Self-Contained

Now that we know how model serialization works in queued jobs, we understand that a fresh model instance will be fetched when the worker picks the job up. The state of that model may have changed since the time the job was dispatched.

We must keep our jobs self-contained; it needs to be able to run without facing any negative side effects due to a state change. To apply this in our challenge, let's pass the value of `current_visitors` while dispatching the job:

```
foreach (Site::all() as $site)
{
    if ($site->current_visitors >= $site->threshold) {
        SendSpikeDetectionNotification::dispatch(
            $site, $site->current_visitors
        );
    }
}
```

With this change, the job's payload will contain an instance of `ModelIdentifier` and the site's current visitors when the spike is detected.

Our job class will now look like this now:

```
class SendSpikeDetectionNotification implements ShouldQueue
{
    use SerializesModels;

    public function __construct(
        private Site $site,
        private int $visitors
    ){}
}
```

And here's the `handle()` method:

```
public function handle()
{
    SMS::send(
        $this->site->owner,
        "Spike detected on {$this->site->name}!
        Current visitors: {$this->visitors}"
    );
}
```

When we send the SMS message, we will use the `visitors` class property instead of the `current_visitors` model attribute. So when users receive the alert, they will get accurate information about their site when the spike was detected.

Processing Uploaded Videos

In the last challenge, we learned the importance of keeping our jobs self-contained and ensuring that the job has everything it needs to run without relying on an external state. In this challenge, we will look at one of the common mistakes people make when trying to make their jobs self-contained.

In our application, the backend receives an uploaded video from the browser, compresses it, and then sends it to storage.

The controller action may look like this:

```
class VideoUploadController{
    public function upload($request) {
        CompressAndStoreVideo::dispatch(
```

```
            $request->file('video')
        );
    }
}
```

We mentioned earlier that Laravel takes a job object, serializes it, and then stores it in the queue. So, if we pass a file upload to the job constructor, PHP will attempt to serialize that file. However, it will fail with the following exception:

```
Serialization of 'Illuminate\Http\UploadedFile' is not allowed.
```

The `UploadedFile` class extends PHP's `SplFileInfo`, which is marked as unserializable by the PHP team. So, to send that video to the queue, we need to serialize it ourselves. One way of doing so is using binary serialization. Instead of passing the uploaded file, we're going to pass its binary representation:

```
$content = app('files')->sharedGet(
    request()->file('video')->getPathname()
);

CompressAndStoreVideo::dispatch($content);
```

Inside the `handle()` method of the job class, we can write that file to disk and start processing it.

Using binary serialization works, but it leads to storing the entire file in our job payload, which increases its size dramatically.

Job Payload Size

The serialized version of our `CompressAndStoreVideo` job will be sent to the queue store. Then, each time a worker picks up the job, it will read the entire payload and unserialize it so the job can run.

A large payload means there'll be a considerable overhead each time the worker picks up the job. Even if the job were released back to the queue immediately due to a rate limiter, the payload would still need to be downloaded, and the job will have to be loaded into our server's memory.

Moreover, we are limited by the maximum payload size of some queue drivers, such as SQS,

which only allows a message payload size of 256 KB.

In addition to all this, if we're using the Redis queue driver, we must keep in mind that our jobs are stored in the Redis instance's memory, and memory is neither unlimited nor cheap. So, we need to be very careful when storing data in memory.

Reducing the Payload Size

When working with queued jobs, we always want to keep our payload as simple and small as possible. Therefore, laravel, by default, takes care of converting a `Model` instance to the simpler `ModelIdentifier` instance to reduce the payload size.

We can use the same pattern with our video processing challenge and only send a reference to the file to the queue:

```
$temporaryFilePath = request()
                    ->file('video')
                    ->store('uploaded-videos');

CompressAndStoreVideo::dispatch($temporaryFilePath);
```

Here we store the file temporarily on disk and send the file path to the job instead of the file itself.

Now inside the job, we can load the video file from the temporary location, process it, store the output in a permanent location, and then delete the temporary file:

```
public function handle()
{
    $newFile = VideoProcessesor::compress(
        $this->temporaryFilePath
    );

    // ...

    app('files')->delete(
        $this->temporaryFilePath
    );
}
```

This pattern is called *Claim Check*. It is analogous to the luggage check at the airport. We

wouldn't want to carry all our luggage around the airport and on the plane. So we check it with the airline and receive a sticker with a reference. Once we reach our destination, we use this sticker to claim our luggage.

Handling Job Failure

Now let's look into how we want to handle the failure of this job. When the job fails, the temporary file will still occupy a space in our local disk. If we are not going to retry the job manually later, we can delete the temporary file using the `failed()` method in our job class:

```
public function failed(Exception $e)
{
    app('files')->delete(
        $this->temporaryFilePath
    );
}
```

However, if we want to retry this job manually, we must keep the temporary file. In that case, we can dispatch a job to delete the temporary file—if it wasn't deleted already—after a reasonable amount of time:

```
public function failed(Exception $e)
{
    DeleteFile::dispatch($this->temporaryFilePath)->delay(
        now()->addHours(24)
    );
}
```

> **Warning:** The Amazon SQS queue service has a maximum delay time of 15 minutes.

Now in the `handle()` method of the `DeleteFile` job, we'll need to check if the temporary file still exists before running the job:

```
public function handle()
{
    if (! app('files')->exists(
        $this->temporaryFilePath
```

```
    )) {
        report(
            new \Exception('Temporary file not found!')
        );

        return $this->delete();
    }
}
```

Here we check if the file doesn't exist and report an exception to be stored in the logs, then we delete the job from the queue so that it's not retried again.

Now the temporary file will be automatically deleted after 24 hours. During this time, we may configure the job to retry automatically, or we can retry the job manually from the dead-letter queue.

Sending Monthly Invoices

Let's look at another confusing concept, which is the concept of shared memory. We're all used to the fact that PHP handles every request in its own separate memory space. But that's not the case for queue workers. Jobs inside a single worker are handled within a shared memory space.

In this challenge, our application sends a monthly email to every user with an invoice showing their monthly usage and the payment amount due. The payment is calculated in USD, and every user gets invoiced with their local currency. Here's what the code in the scheduled command looks like:

```
foreach (User::all() as $user) {
    GenerateInvoice::dispatch($user, $month);
}
```

Inside the `GenerateInvoice` job, we calculate the usage, convert it to the local currency, add taxes, create an invoice, and finally email it to the user:

```
public function handle()
{
    // Amount due in USD
    $amount = UsageMeter::calculate($this->user, $this->month);
```

```
    $amountInLocalCurrency = $amount * Currency::exchangeRateFor(
        $this->user->currency
    );

    $taxInLocalCurrency = ($amount * 14 / 100) * Currency::exchangeRateFor(
        $this->user->currency
    );

    $total = $amountInLocalCurrency + $taxInLocalCurrency;

    Mail::to($this->user)->send("Your usage last month was
        {$total}{$this->user->currency}"
    );
}
```

Inside the `Currency` class, we send an API request to find the current exchange rate and store it in a static variable. This makes us avoid sending that request each time we want to convert to the same currency:

```
class Currency{
    public static $rates = [];

    public static function exchangeRateFor($currency)
    {
        if (!isset(static::$rates[$currency])) {
            static::$rates[$currency] = ExchangeRateApi::get(
                'USD', $currency
            );
        }

        return static::$rates[$currency];
    }
}
```

> **Notice:** This technique is called "Memoization".

When we call `Currency::exchangeRateFor('CAD')`, the currency converter will use the `ExchangeRateApi` service to get the USD-to-CAD exchange rate and store it in the `$rates` static property.

Next time we call `Currency::exchangeRateFor()` to convert the tax, the converter will

not use the `ExchangeRateApi` service again. Instead, it'll use the rate stored inside the `$rates` property.

Using the Same State

The `GenerateInvoice` job will work perfectly for the first month. In the following month, users will start reporting the numbers are wrong. This is because the exchange rate used in the calculations is incorrect.

Even though a brand-new set of jobs is dispatched each month, the same exchange rate from the previous month was used. And the reason is that the `Currency::$rates` static property had the old rates stored. So, when `Currency::exchangeRateFor()` was called on the following month, the converter didn't use the `ExchangeRateApi` service to get the fresh data.

When we start a worker process, a single application instance is booted up and stored in memory. All jobs processed by this worker will be using that same application instance and the same shared memory.

Making Our Job Self-Contained

It's always a good practice to ensure queued jobs are self-contained; that each job has everything it needs to run. So, while dispatching the jobs, we're going to get a new exchange rate and then pass it to each job:

```
foreach (User::all() as $user) {
    $exchangeRate = Currency::exchangeRateFor(
        $user->currency
    );

    GenerateInvoice::dispatch(
        $user, $month, $exchangeRate
    );
}
```

> **Notice:** Dispatching the jobs is happening inside a scheduled CRON task. Each time the task runs, a new state is created for it.

Inside the `handle()` method of the `GenerateInvoice` job, we're going to use that

exchange rate passed to the constructor:

```
public function handle()
{
    // ...

    $amountInLocalCurrency = $amount * $this->exchangeRate;

    $taxInLocalCurrency = ($amount * 14 / 100) * $this->exchangeRate;

    // ...
}
```

Now the most recent exchange rate will be fetched—each month—and used when the jobs are being processed.

A Fresh State for Each Job

While every HTTP request and every console command is handled using a fresh application instance, queues run in a single process. Why? Because it gives us better performance.

Creating a new process on our machine and booting up the Laravel application—with all service providers, packages, and container bindings—consumes valuable resources and time. However, creating the process once allows your application to run more jobs faster and with less overhead.

However, if you need to have a fresh instance each time, you can use the `queue:listen` command instead:

```
php artisan queue:listen --timeout=60 --backoff=30,60 --tries=2
```

This command starts a long-living process on your server; the difference between this process and the worker process is that it will begin with a new child process for each job in your queue.

That child process will have a fresh application instance and a new state.

Controlling The Rate of Job Execution

When building an infrastructure for dispatching and processing queued jobs, we'd generally focus on ensuring the jobs are processed as soon as possible, and the queue is emptied as quickly as possible. However, in some specific cases, the business logic may force us to limit the rate of job execution.

In this challenge, we'll look into a business intelligence web application that allows customers to generate on-demand reports. But each customer can only have five reports generated at the same time. That way, the system distributes its resources equally between customers. No customer will block all available workers while generating reports for themselves.

Here's how the job looks:

```
public function handle()
{
    $results = ReportGenerator::generate(
        $this->report
    );

    $this->report->update([
        'status' => 'done',
        'results' => $results
    ]);
}
```

Report generation takes a few minutes to complete, and the results are stored in the `Report` model.

Funnelling Jobs

We want only to allow five jobs belonging to a single customer to run simultaneously. Any more report generation requests from the same customer shall be delayed until one of the five slots is freed.

Luckily, Laravel ships with its concurrency limiter that we can easily implement in any of our queued jobs:

```
public function handle()
{
    Redis::funnel($this->report->customer_id)
    ->limit(5)
    ->then(function () {
        $results = ReportGenerator::generate(
            $this->report
        );

        $this->report->update([
            'status' => 'done',
            'results' => $results
        ]);
    }, function () {
        return $this->release(10);
    });
}
```

Using `Redis::funnel()`, we've limited the execution of the report generation to only five concurrent executions.

> **Notice:** We've used the `customer_id` as the lock key to apply the limit per customer.

If we could acquire a lock on one of the five slots, the limiter would execute the first closure passed to the `then()` method, which is the business logic for the report generation. However, if acquiring a lock wasn't possible, the second closure will be executed, and the job will be released back to the queue to be retried after 10 seconds.

> **Notice:** To use the rate limiters, you need to configure a Redis connection in your `database.php` configuration file.

> **Notice:** Your queue connection doesn't have to use the Redis driver to be able to use the rate limiters.

Waiting a Few Seconds

By default, the limiter will wait 3 seconds before it gives up and releases the job. You can control the wait time by using the `block` method:

```
Redis::funnel($this->report->customer_id)
    ->block(5)
    ->limit(...)
    ->then(...)
```

`block(5)` will instruct the limiter to keep trying to acquire a lock for 5 seconds.

Setting a Timeout for a Slot

By default, the limiter will force evacuate a slot after 60 seconds. If our report generation task takes more than 60 seconds, a slot will be freed, and another job may start, causing six concurrent jobs to run. We don't want to allow that.

Let's configure the timeout to be 5 minutes instead. We can do that using the `releaseAfter()` method:

```
Redis::funnel($this->report->customer_id)
    ->releaseAfter(5 * 60)
    ->block(...)
    ->limit(...)
    ->then(...)
```

Each slot will be freed once a report generation is completed or if 5 minutes have passed.

Managing Tries

Since jobs can be released back to the queue several times, we'll need to limit the number of attempts by setting `$tries` on the job class:

```
public $tries = 10;
```

However, if `ReportGenerator` throws an exception, we don't want to keep retrying ten times. So, we're going to limit the allowed exceptions to only 2:

```
public $tries = 10;
public $maxExceptions = 2;
```

Now the job will be attempted ten times. Two of those attempts could be due to an exception being thrown or the job timing out.

Limiting the Job Rate

Now let's explore how we can rate-limit generating reports to only five reports per hour. Instead of using `Redis::funnel()`, we're going to use another built-in limiter called `Redis::throttle()`:

```
public function handle()
{
    Redis::throttle($this->report->customer_id)
        ->allow(5)
        ->every(60 * 60)
        ->then(function () {
            $results = ReportGenerator::generate(
                $this->report
            );

            $this->report->update([
                'status' => 'done',
                'results' => $results
            ]);
        }, function () {
            return $this->release(60 * 10);
        });
}
```

In the example above, we're allowing five reports to be generated per hour. If all slots were occupied, we will release the job to be retried after 10 minutes.

The hour starts counting from the time the first slot was occupied. So, for example, if the first report was initiated at 10:30:30, the limiter will reset at 11:30:30.

Dealing With API Rate Limits

If an application communicates with a third-party API, there's a big chance some rate-limiting strategies will be applied. Let's see how we may deal with a job that sends an HTTP request to an API that only allows 30 requests per minute:

Here's how the job may look:

```
public $tries = 10;

public function handle()
{
    $response = Http::acceptJson()
        ->timeout(10)
        ->get('https://...');

    if ($response->failed() && $response->status() == 429) {
        return $this->release(30);
    }

    // ...
}
```

If we get a rate limit response `429 Too Many Requests` from the service, we release the job back to the queue to be retried again after 30 seconds. We also configured the job to be attempted ten times by setting the `$tries` class property to 10.

Not Sending Requests That We Know Will Fail

When we reach the service's rate limit, any request sent will be blocked. So, if we know requests will keep failing, there's no point in sending them and delaying processing other jobs in the queue. Instead, we need a mechanism to prevent processing jobs that communicate with this service until the blockage is lifted.

To do that, we will set a key in the cache when we hit the limit and immediately release the job if the key hasn't expired. Typically when an API responds with a 429 response code, a `Retry-After` header will be sent with the number of seconds remaining until we can send requests again:

```
if ($response->failed() && $response->status() == 429) {
    $secondsRemaining = $response->header('Retry-After');
```

```
        Cache::put(
            'api-limit',
            now()->addSeconds($secondsRemaining)->timestamp,
            $secondsRemaining
        );

        return $this->release(
            $secondsRemaining
        );
    }
```

Here we inspect the `Retry-After` header, extract the seconds remaining until the blockage is lifted, and store it in a `$secondsRemaining` variable. Then, we set an `api-limit` cache key with the timestamp of when requests will be allowed again. We also put this cache key to expire after the number of seconds stored in `$secondsRemaining` has passed.

Finally, we're going to use the value of the `$secondsRemaining` variable to set the delay for the released job:

```
return $this->release(
    $secondsRemaining
);
```

That way, the job will be available as soon as requests are allowed again.

> **Warning:** When dealing with input from external services—including headers—it might be a good idea to validate that input before using it.

Now we're going to check for that cache key at the beginning of the `handle()` method of our job and release the job back to the queue if the cache key hasn't expired yet:

```
public function handle()
{
    if ($timestamp = Cache::get('api-limit')) {
        return $this->release(
            $timestamp - time()
        );
    }

    // ...
```

```
}
```

`$timestamp - time()` will give us the seconds remaining until requests are allowed.

Here's the whole thing:

```
public function handle()
{
    if ($timestamp = Cache::get('api-limit')) {
        return $this->release(
            $timestamp - time()
        );
    }

    $response = Http::acceptJson()
        ->timeout(10)
        ->get('https://...');

    if ($response->failed() && $response->status() == 429) {
        $secondsRemaining = $response->header('Retry-After');

        Cache::put(
            'api-limit',
            now()->addSeconds($secondsRemaining)->timestamp,
            $secondsRemaining
        );

        return $this->release(
            $secondsRemaining
        );
    }

    // ...
}
```

> **Notice:** In this part of the challenge, we only handle the 429 request error. In the actual implementation, you'll also need to address other 4xx and 5xx errors.

Avoiding a Thundering Herd

When we hit a rate limit response, we release all jobs back to the queue to be retried once the blockage is lifted. That means the moment the blockage is lifted, hundreds—or maybe thousands—of jobs will flood the queue and fill it up completely. This herd of jobs will keep your workers busy for a long time, which may delay processing other jobs in the queue. Moreover, the burst of traffic to the service will possibly make us hit the rate limit again.

This problem is called the *thundering herd* problem, it was mainly used in the context of operating system threads/processes responding to system events, but it can also apply to multiple jobs of the exact nature, suddenly becoming available in the queue.

A solution to this problem is to add some randomness to the release delay; this is called *jitter* in architectural terms. Using a jitter allows us to spread the spikes over a relatively long period.

To not overcomplicate things, we can implement this concept by adding a random number of seconds to each delay:

```
return $this->release(
    $secondsRemaining + rand(0, 300)
);
```

Now the jobs will be retried up to 5 minutes after the blockage is lifted.

Replacing the Tries Limit with Expiration

Since the request may be throttled multiple times, it's better to use the job expiration configuration instead of setting a static tries limit.

```
public $tries = 0;

// ...

public function retryUntil()
{
    return now()->addHours(12);
}
```

Now, if the limiter throttled the job multiple times, it will not fail until the 12-hour period passes.

Limiting Exceptions

If an unhandled exception was thrown from inside the job, we don't want it to keep retrying for 12 hours. For that reason, we're going to set a limit for the maximum exceptions allowed:

```
public $tries = 0;
public $maxExceptions = 3;
```

Now, the job will be attempted for 12 hours but will fail immediately if three attempts fail due to an exception or a timeout.

Dealing With an Unstable Service

In the previous challenge, we implemented a mechanism to stop communicating with a service that is rate limited when we hit the limit. Doing so saved us valuable system resources that we may allocate to processing other jobs.

In this challenge, we will look into dealing with a service that isn't stable. HTTP requests to their API respond with a 500 internal server error, and it can take hours before the service goes back up.

We want to minimize the impact on our system when that service is down. So, we typically want to stop sending requests to the service if we know it has been down for a while.

Here's how our job may look:

```
public function handle()
{
    $response = Http::acceptJson()
        ->timeout(10)
        ->get('https://...');

    if ($response->serverError()) {
        return $this->release(600);
    }

    // Use the response to run the business logic.
}
```

Implementing a Circuit Breaker

What we want to implement here is called "The Circuit Breaker" pattern. It's a way to prevent failure from constantly recurring. Once we notice a particular service is failing multiple consecutive times, we'll stop sending requests to it for a while and give it some time to recover.

Laravel ships with a circuit breaker implementation in the form of a job middleware. Like the HTTP middleware, Laravel allows us to run our job inside a middleware pipeline. The one we want to use here is the `ThrottlesExceptions` middleware.

To register the middleware inside our job, we need to add a `middleware` method that returns an array:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

public function middleware()
{
    return [new ThrottlesExceptions];
}
```

This middleware catches exceptions thrown inside the job and increments a counter. When a certain number of failures is noticed, the middleware will start releasing the job back to the queue without actually running it. And thus, prevent the queue from attempting to communicate with this unstable service while it's down. After a while, the counter resets and the job is allowed to run again.

Figure 2-25 shows a logical view of the circuit breaker. During the first two executions of the job, the worker checked with the circuit breaker and found it closed. So, the circuit breaker allowed the request to the service, but both requests failed. After the second attempt failed, the circuit breaker trips and opens the circuit.

When the third job started, the circuit was open, so no request was sent to the service, and the job was released back to the queue immediately.

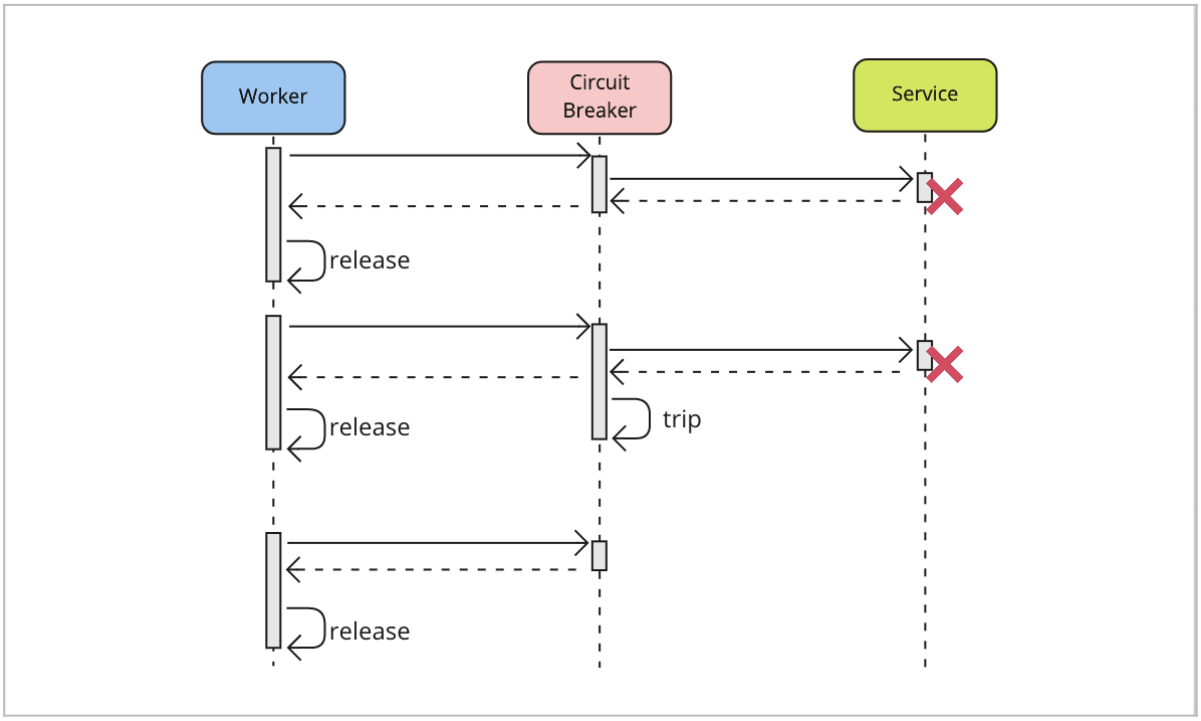

Figure 2-25. A circuit breaker.

> **Notice:** I studied electrical and mechanical engineering for a few years, and it still confuses me a bit that an open circuit means **no flow**. A closed circuit means the opposite. (Open = don't send requests, Closed = send).

In our example, we're using Laravel's HTTP client. Which doesn't throw exceptions by default. We want to change this behavior, so our circuit breaker works. For that, we may use the `throw()` method:

```
$response = Http::acceptJson()
                ->timeout(10)
                ->get('https://...')
                ->throw();
```

Now the client will throw an `Illuminate\Http\Client\HttpClientException` exception when the request fails.

Configuring the Circuit Breaker

By default, the `ThrottlesExceptions` middleware will allow ten failures before opening the circuit. It also keeps the circuit open for 10 minutes before closing it. To change these

defaults, we can provide our values to the middleware constructor:

```
public function middleware()
{
    return [
        new ThrottlesExceptions(
            maxAttempts: 3,
            decayMinutes: 5
        )
    ];
}
```

Now the circuit breaker will only allow three failures and keep the circuit open for 5 minutes.

Another thing we want to configure is the scope of the circuit breaker. For example, should it count failures on the job level? or across all jobs of the same instance? or across the entire queue covering all jobs that communicate with this service?

This is determined by the cache key Laravel assigns for the circuit breaker; by default, it's the name of the job class. That means all instances of this job class will go through this service breaker.

In our case, we want all jobs that communicate with this service to go through the service breaker. For that reason, we will give the circuit breaker a unique name that we'll use when the middleware is applied to all those jobs. And to do that, we'll use the `by()` method on the middleware:

```
public function middleware()
{
    return [
        (new ThrottlesExceptions(
            maxAttempts: 3,
            decayMinutes: 5
        ))->by('service_id_here');
    ];
}
```

Now let's configure the number of minutes a released job will be delayed when the circuit breaker counts a failure. In figure 2-25, you can see the jobs are released during the first two attempts; that's what I mean here. And to configure that, we may use the `backoff()` method of the `ThrottlesExceptions` middleware:

```
public function middleware()
{
    return [
        (new ThrottlesExceptions(
            maxAttempts: 3,
            decayMinutes: 5
        ))->by('service_id_here')
          ->backoff(1);
    ];
}
```

Now the job will be released with a 1-minute delay.

Defining The Exception

By default, the circuit breaker will catch any exception thrown from within our job. However, we want to activate the circuit breaker only when an exception is thrown from our HTTP client, not any other exception. And to do that, we may use the `when()` method:

```
public function middleware()
{
    return [
        (new ThrottlesExceptions(
            maxAttempts: 3,
            decayMinutes: 5
        ))->by('service_id_here')
          ->backoff(1)
          ->when(function (Throwable $e){
              return $e instanceof HttpClientException
          });
    ];
}
```

Now the circuit breaker will only get activated when the exception thrown within the job is an instance of `HttpClientException`, which indicates a failure message or a connection error.

Using Redis

The `ThrottlesExceptions` middleware uses Laravel's cache component by default to store the circuit breaker data. If you have Redis configured in your system, it might be a

good idea to use it for the circuit breaker instead of the default cache driver.

To use Redis, switch to the `ThrottlesExceptionsWithRedis` instead. This middleware is fine-tuned for Redis and thus offers much better performance.

Bulkheading

The circuit breaker we put in place saved us valuable system resources. When the service is having issues, we won't waste time trying to reach it. However, there's another issue that we need to deal with, which is the case when the service doesn't respond at all.

We already have a connection timeout of 10 seconds set on the HTTP client:

```
$response = Http::acceptJson()
    ->timeout(10)
    ->get('...');
```

However, depending on the number of workers running, it'll take at least 10 seconds for the circuit breaker to trip if the service is not responding.

Why 10 seconds? If we have multiple workers all concurrently processing a job that calls this service, each job will take 10 seconds before the HTTP request timeouts, and the circuit breaker counts the failure. During those 10 seconds, our workers will wait for the timeout requests. No other jobs will be processed.

To avoid that, we may dedicate a limited number of workers to consume jobs that communicate with this service. So, for example, we dispatch the job to a `limited` queue and start five workers with these configurations:

```
php artisan queue:work --queues=limited,default
```

The other five workers with the following configurations:

```
php artisan queue:work --queues=default
```

That way, a maximum of 5 workers could be stuck processing jobs from the `limited` queue while other workers are processing other jobs.

Another way of dealing with this issue is by using the concurrency limiter that comes with

Laravel:

```
Redis::funnel('slow_service')
    ->limit(5)
    ->then(function () {
        // ...

        $response = Http::acceptJson()
        ->timeout(10)
        ->get('...');

        // ...
    }, function () {
        return $this->release(10);
    });
```

With the limiter configurations above, only five workers will be running this job simultaneously. If another worker picks that job up while all five slots are occupied, it will release it immediately. Giving a chance for other jobs to be processed.

Limiting the possibility of failure to only five workers is a simple form of a pattern known as Bulkheading. Combined with a circuit breaker, it leads to an efficient fault-tolerant system.

Provisioning a Serverless Database

In this challenge, we want our users to be able to provision a serverless database on the AWS cloud platform. We need to have a network, give that network internet access, and then create the database.

Each of these steps may take up to 5 minutes to complete. AWS will start the process with a `provisioning` state, and we'll have to monitor the resource until the state becomes `active`. We can't run a step until the resource from the preceding step is `active`.

Instead of writing this logic in one complicated queued job and dispatching it to the queue, let's put each of the three steps in a separate job:

- `EnsureNetworkExists`
- `EnsureNetworkHasInternetAccess`
- `CreateDatabase`

Inside the `handle` method of our `EnsureNetworkExists`, we will check if a network

doesn't exist yet, create one and wait for it to become active (Figure 2-26).

It waits by releasing the job back to the queue to be processed later.

```
public function handle()
{
    $network = Network::first();

    if (!$network) {
        Network::create(...);

        return $this->release(120);
    }

    if ($network && $network->isActive()) {
        return;
    }

    return $this->release(120);
}
```

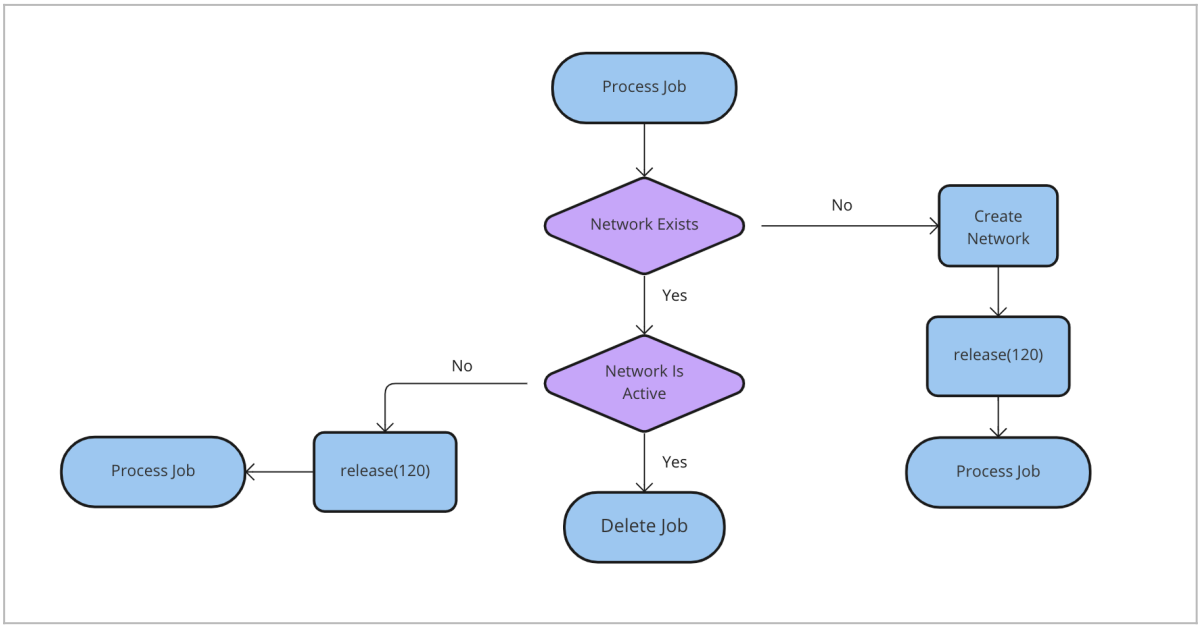

Figure 2-26. The business logic inside EnsureNetworkExists.

The `EnsureNetworkHasInternetAccess` job works similarly. Releases the job back several times until the network has internet access.

```
public function handle()
{
```

```
    $network = Network::first();

    if ($network->hasInternetAccess()) {
        return;
    }

    $network->addInternetAccess();

    return $this->release(120);
}
```

Finally, the `CreateDatabase` job creates the database instance by attaching it to the network and monitors its status until it becomes active (Figure 2-27).

```
public function handle()
{
    $database = $this->database;
    $network = Network::first();

    if (! $database->isAttached()) {
        $network->attachDatabase($database);

        return $this->release(120);
    }

    if ($database->isActive()) {
        return;
    }

    return $this->release(120);
}
```

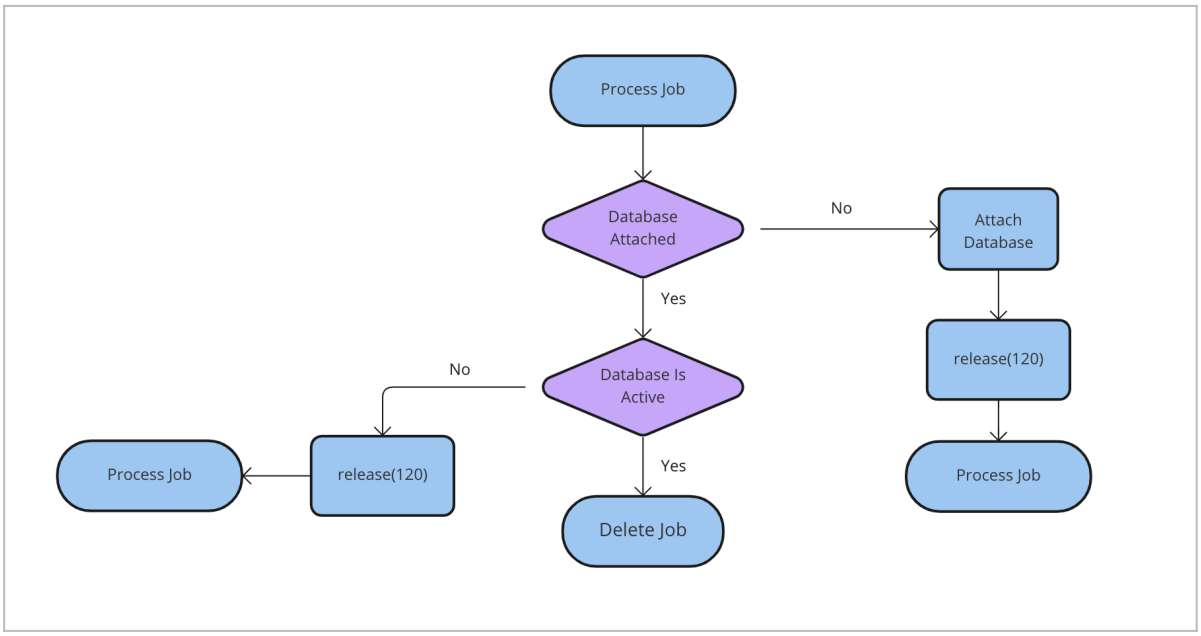

Figure 2-27. The business logic inside CreateDatabase.

These three jobs need to run one after the other. We also need to clean the resources created in case any steps fail. You can think of it as a database transaction where all the updates are committed successfully, or everything is rolled back to the original state.

Dispatching a Chain of Jobs

Laravel's command bus provides a method for dispatching a chain of jobs. Here's how we can use this to dispatch our jobs:

```
$database = Database::create([
    'status' => 'provisioning'
]);

Bus::chain(
    new EnsureNetworkExists(),
    new EnsureNetworkHasInternetAccess(),
    new CreateDatabase($database)
)->dispatch();
```

First, we will create a record in the database with the status `provisioning`. Then, we dispatch the chain using `Bus::chain` and pass the database model to the `CreateDatabase` job.

Unlike batches, jobs inside a chain are processed in sequence. So if a job didn't report completion, it'd be retried until it either succeeds, and the next job in the chain is dispatched

or fails, and the whole chain is deleted.

Figure 2-28 shows how a chain works. Only if the `EnsureNetworkExists` completes, the `EnsureNetworkHasInternetAccess` job is dispatched to the queue, and only if it completes the `CreateDatabase` job is dispatched. If any jobs fail, they will be retried and either complete or move to the dead-letter queue.

When dispatching job batches, all the jobs are sent to the queue in one go and can be executed concurrently. With chains, a job is only dispatched to the queue when the job preceding it completes.

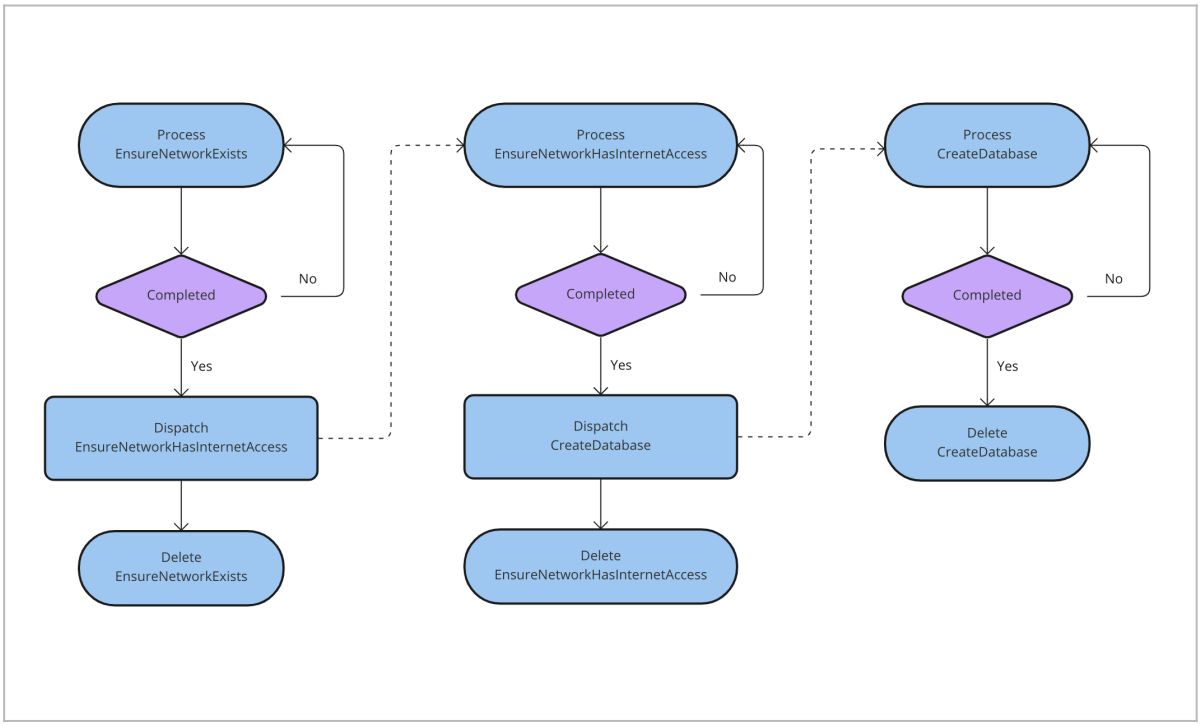

Figure 2-28. The chain dispatches jobs one after the other.

Handling Failure

In case of failure, we need a way to roll back any changes that might have occurred while trying to create the database. In some cases, that may mean deleting all resources created. In our case, it's a bit more complicated. Before removing internet access from the network, we need to ensure any other resource does not use it. Same with removing the network entirely.

This process of intelligently undoing the effect of a chain of jobs is called a "*Compensating Transaction*".

Here's how it may look:

```
$network = Network::first();

if (! $network->usedByOtherResources()) {
    $network->delete();

    return;
}

if (! $network->activeDatabases()->count()) {
    $network->removeInternetAccess();
}
```

First, we check if the network isn't attached to any other resources, which means it's safe to delete it. However, if it is used by other resources but has no active databases, we remove the internet access while keeping the network.

To run this compensating transaction, we can attach it to the chain using `catch()`:

```
Bus::chain(
    new EnsureNetworkExists(),
    new EnsureNetworkHasInternetAccess(),
    new CreateDatabase($database)
)->catch(function() {
    $network = Network::first();

    if (! $network->usedByOtherResources()) {
        $network->delete();

        return;
    }

    if (! $network->activeDatabases()->count()) {
        $network->removeInternetAccess();
    }
})->dispatch();
```

If any job in the chain fails, the closure passed to `catch()` will be invoked, and the compensating transaction will run.

> **Notice:** When I say a job *failed*, I mean it has consumed all configured attempts. In other words,`catch()` will not be invoked until all configured attempts of a job are exhausted.

> **Notice:** The `failed()` method of a failed job will also be triggered after the closure from the `catch()` method is invoked.

Minding Timeouts

If you have a timeout set for a chain job that calls a `catch()` callback, the time spent on making this call will be counted on the job. In other words, the job may timeout while in the middle of calling these callbacks.

When this happens, the job may be retried while it has already completed. This is similar to calling the callbacks when processing a job that's part of a batch.

So, we should keep the tasks running inside `catch()` as short as possible. If we need to perform some lengthy work, we can dispatch another queued job that has its own timeout:

```
Bus::chain(
    // ...
)->catch(function() {
    $network = Network::first();

    RemoveUnusedNetworkResources::dispatch($network);
})->dispatch();
```

Generating Complex Reports

The size and structure of data can make generating reports very complicated and time-consuming. Running this task in the background will allow us to respond faster to the user and utilize our resources more efficiently.

Given a large spreadsheet, this challenge is about reading the data, transforming it into a simpler form that we can query, storing that new form, and then generating the final summary.

To keep our jobs short, we're going to have a separate job for each step:

- ExtractData
- TransformChunk
- StoreData
- GenerateSummary

Chaining and Batching

While extracting data from the spreadsheet, we will split it into smaller chunks. This will allow us to work on these chunks in parallel and thus generate the report faster.

So, the `ExtractData` job is going to transform the spreadsheet into multiple chunks, and then we can dispatch a batch with all the `TransformChunk` jobs:

```
$report = Report::create([...]);

Bus::chain([
    new ExtractData($report),
    function () use ($report) {
        $jobs = collect($report->chunks)->mapInto(
            TransformChunk::class
        );

        Bus::batch($jobs)->dispatch();
    }
])->dispatch();
```

Here we created a report model that holds details about the progress of the report generation process, and we pass that model to the `ExtractData` job dispatched in a chain. Once the `ExtractData` job finishes, Laravel will dispatch the next job in the chain. In that case, instead of a job object, we will dispatch a closure.

When the closure runs, it will collect the chunks created in the extraction step and dispatches a batch of `TransformChunk` jobs. Each job will receive the chunk as a constructor parameter and will be responsible for transforming the data in this chunk.

Dispatching a Chain After the Batch Finishes

Now, after the batch runs, we need to collect the parts created by the multiple `TransformChunk` jobs and store them in our database. Only then we can generate a summary out of this data:

```
$report = Report::create([...]);

Bus::chain([
    new ExtractData($report),
    function () use ($report) {
        $jobs = $report->chunks->mapInto(
            TransformChunk::class
        );

        Bus::batch($jobs)->then(function() use ($report) {
            Bus::chain([
                new StoreData($report),
                new GenerateSummary($report)
            ])->dispatch();
        })->dispatch();
    }
])->dispatch();
```

Here we're using the `then()` method from the batch and dispatching a chain that runs `StoreData` and `GenerateSummary` in sequence.

To give you a clearer image of what we built, here's an execution plan (Figure 2-29):

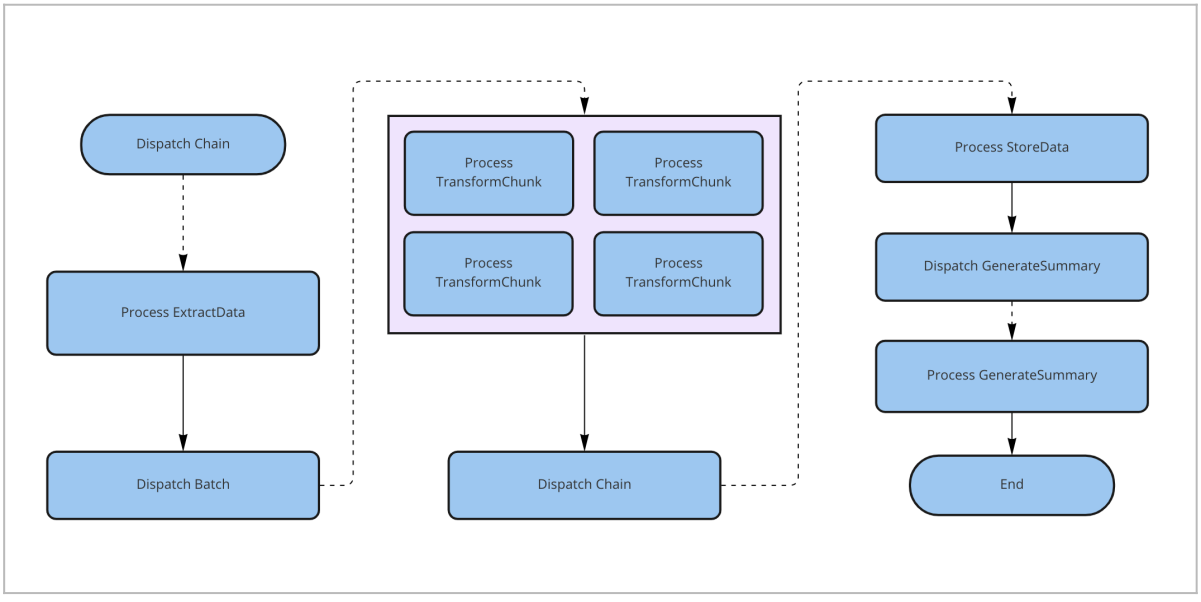

Figure 2-29. The execution plan of the workflow.

Making It More Readable

The code from above can be a little confusing. And if the process is more complex, it will be very confusing.

To make it easier to read, let's create a `ReportGenerationPlan` class:

```
class ReportGenerationPlan{
    public static function start($report)
    {
        Bus::chain([
            new ExtractData($report),
            function () use ($report) {
                static::step2($report);
            }
        ])->dispatch();
    }

    private static function step2($report)
    {
        $jobs = $report->chunks->mapInto(
            TransformChunk::class
        );

        Bus::batch($jobs)
        ->then(function() use ($report) {
            static::step3($report);
        })
        ->dispatch();
    }

    private static function step3($report)
    {
        Bus::chain([
            new StoreData($report),
            new GenerateSummary($report)
        ])->dispatch();
    }
}
```

> **Warning:** Always use static methods in your plan classes to avoid using `$this` inside closures. The serialization of a closure that contains `$this` is buggy.

This class represents our execution plan with its three steps:

- Dispatching a chain to extract the data
- Dispatching a batch to transform all chunks

- Dispatching a chain to store the results and generate a summary

To start executing the plan, we'll only need to call `start()`:

```
$report = Report::create([...]);

ReportGenerationPlan:::start($report);
```

Handling Failure

In case of failure, we might want to clean any unnecessary files or temporary data. To do that, we can add a new static method to the `ReportGenerationPlan`:

```
private static function failed($report)
{
    // Run any cleaning work ...

    $report->update([
        'status' => 'failed'
    ]);
}
```

We can then call that method from inside the `catch()` closures for the chains and batches we dispatch:

```
class ReportGenerationPlan{
    public static function start($report)
    {
        Bus::chain([
            ...
        ])
        ->catch(function() use ($report) {
            static::failed($report);
        })
        ->dispatch();
    }

    private static function step2($report)
    {
        // ...
```

```
        Bus::batch($jobs)
        ->then(...)
        ->catch(function() use ($report) {
            static::failed($report);
        })
        ->dispatch();
    }

    private static function step3($report)
    {
        Bus::chain([
            ...
        ])
        ->catch(function() use ($report) {
            static::failed($report);
        })
        ->dispatch();
    }

    private static function failed($report)
    {
        // Run any cleaning work ...

        $report->update([
            'status' => 'failed'
        ]);
    }
}
```

FIFO Queues

Laravel queues operate on a first-in, first-out (FIFO) principle. This means that the jobs are picked up for processing in the order they arrive. However, as we have seen multiple times in this book, a job can be released to the back of the queue several times before actually processing it. This means there's no guarantee that jobs inside a queue are processed in the same order they arrive.

In some systems, though, processing jobs must happen in the same order they were dispatched. For example, think of a messaging application that aggregates messages from different social media platforms and aggregates them into a customer support platform. In such a system, we must ensure messages are delivered in the correct order.

If we dispatch a job for each message the app receives, there's no guarantee our workers

will pick those messages up in the correct order. For example, while message #1 was being processed, the support platform experienced an issue, so the job was released back to the queue. Meanwhile, message #2 came, and its job was processed and sent successfully before the job for message #1 was retried.

To simplify our challenge, let's assume we only receive messages from Telegram and send them to Intercom. We want to make sure messages to the same customer are delivered in the correct order without delaying messages to other customers.

Figure 2-30 illustrates a situation where messages came to the queue in the correct order. However, the job for message #1 failed to process, so it was released back to the queue. We want this job to be put at the back of the queue along with all messages to that customer, so we can send the messages in the correct order.

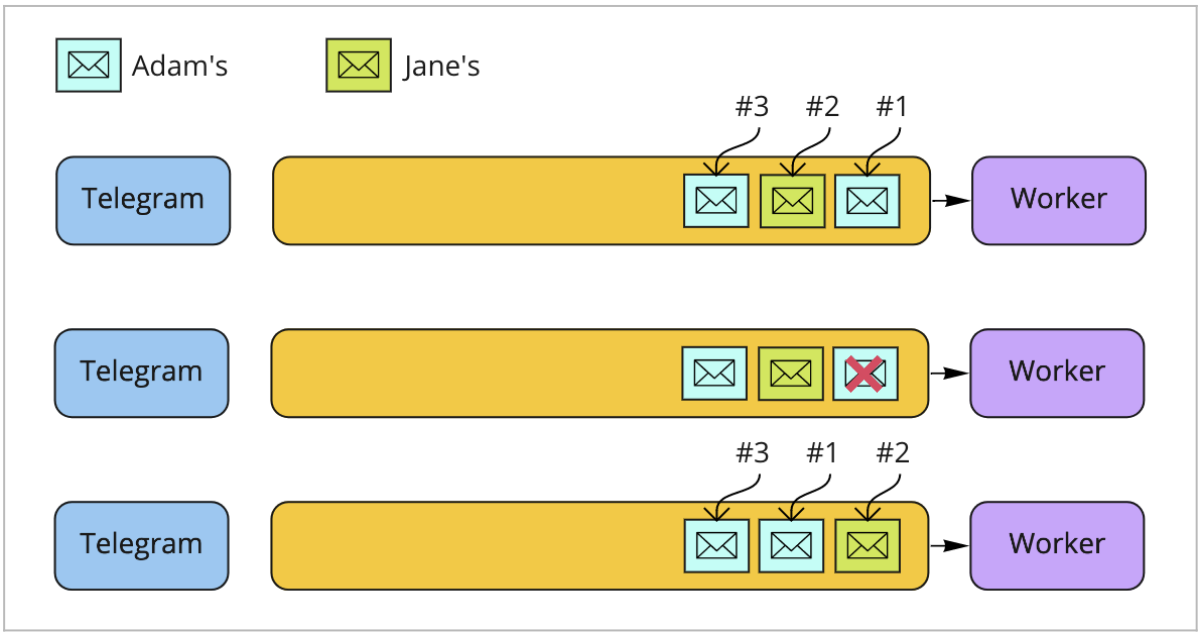

Figure 2-30. Processing messages of two different customers.

Secondary Queue

To achieve this, we're going to create a secondary queue. This queue is a database table that stores all incoming messages in the order they arrive. We can group these messages by the customer and sort them using a timestamp:

```
{
    "message": "Message #1 to Adam",
    "customer": "Adam",
    "timestamp": "1593492111"
}
```

```
{
    "message": "Message #2 to Jane",
    "customer": "Jane",
    "timestamp": "1593492166"
}

{
    "message": "Message #3 to Adam",
    "customer": "Adam",
    "timestamp": "1593492150"
}
```

To find the messages in the queue for a particular customer, we can run a query like this:

```
SELECT * FROM messages WHERE customer = "Adam" ORDER BY timestamp
```

Dispatching a Messenger Job

In previous challenges, we learned that Laravel ships with an API to dispatch job chains. Jobs in a chain are guaranteed to run in the order in which they're added. This is perfect for our use case.

However, we need to find a way to query the `messages` table for each customer and push jobs in a chain dedicated to the customer's jobs only. A chain for Adam's jobs (#1 & #3) may look like this:

```
Bus::chain([
    new ProcessMessage('message #1'),
    new ProcessMessage('message #3'),
    // ...
])->dispatch();
```

To do this, we will start by pushing a regular job once a customer account is activated. This job will act as a dedicated messenger for the customer that keeps an eye on the `messages` database table and pushes the jobs to the customer's dedicated chain. We'll name this job `Messenger`:

```
// On customer activation

Messenger::dispatch($customer);
```

Here's how the `handle()` method of the `Messenger` job looks:

```
public function handle()
{
    $messages = Messages::where('customer', $this->customer->id)
                ->where('status', 'pending')
                ->orderBy('timestamp')
                ->get();

    if (! $messages->count()) {
        return $this->release(5);
    }

    foreach ($messages as $message) {
        // Put each message in the chain...
         $this->chained[] = $this->serializeJob(
            new ProcessMessage($message)
        );
    }

    // Put another instance of the messenger job at
    // the end of the chain
    $this->chained[] = $this->serializeJob(
        new self($this->customer)
    );
}
```

When the `Messager` job runs for a customer, it will collect messages for that customer and put them in a chain under the `Messenger` job. We do that by serializing the jobs and adding them to the `chained` property of the `Messenger` job class. This property is part of the `Queueable` trait used by all Jobs in Laravel.

After adding all messages to the chain, the job will put a new instance of itself at the end of the chain so that the job runs again after processing all jobs.

Figure 2-31 shows the execution plan for the chain belonging to the application customer Adam. The `Messenger` job adds 2 `ProcessMessage` job to the chain and then adds another instance of `Messenger`. After the workers process both the `ProcessMessage` jobs, the `Messenger` job is dispatched, and the operation starts again.

When no messages are found, the `Messenger` job is released back to the queue to be retried after 5 seconds.

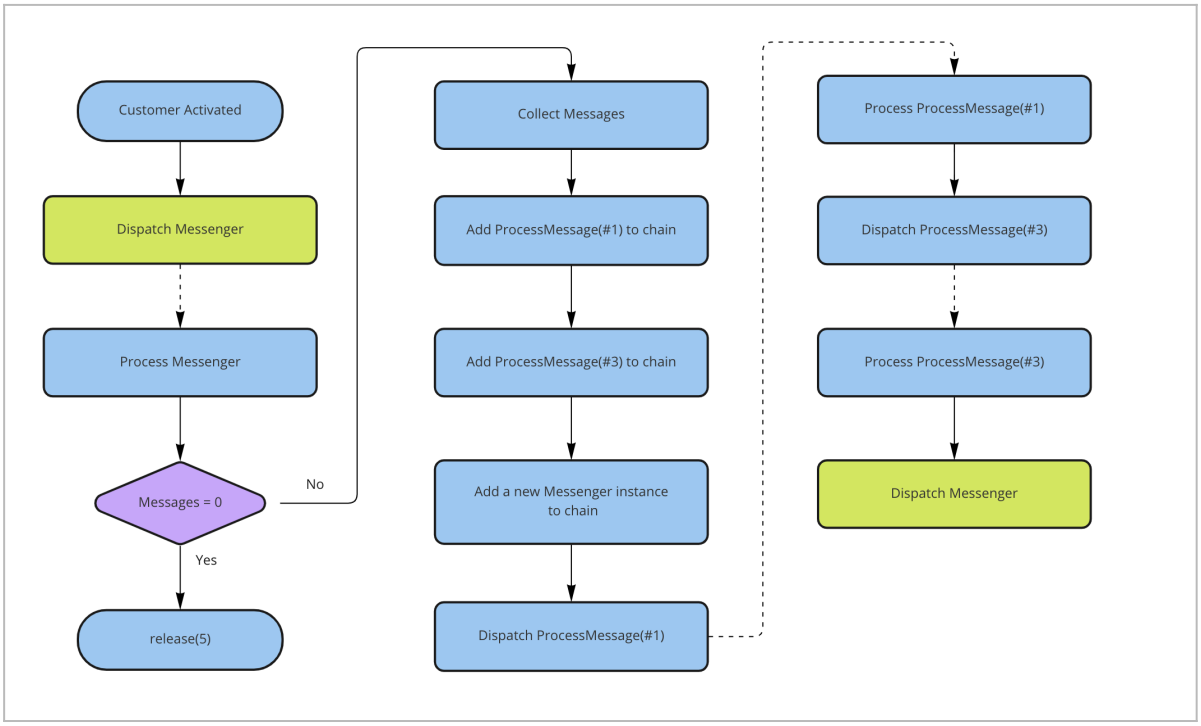

Figure 2-31. Our execution plan.

> **Notice:** Make sure you set `$tries = 0` on the `Messenger` job so it may be released back to the queue an unlimited number of times. Otherwise, it will fail after a few rounds and won't process messages.

This chain will stay alive indefinitely, looking for messages to process for this customer. In addition, other instances of the job may also be running to collect and process messages for other customers.

Sending Messages

Inside the `ProcessMessage` job, the `handle()` method may look like this:

```
public function handle()
{
    Intercom::send($this->message);

    $this->message->update([
        'status' => 'sent'
    ]);
}
```

This job has to be fault-tolerant; if the job is marked as failed, the next jobs in the chain will never run, and the customer will stop receiving new messages. For that reason, we need to let the job retry indefinitely and handle the failure ourselves:

```
public $tries = 0;

public function handle()
{
    if ($this->attemps() > 5) {
        // Log the failure so we can investigate
        Log::error(...);

        // Return immediately so the next job in the chain is run.
        return;
    }

    Intercom::send($this->message);

    $this->message->update([
        'status' => 'sent'
    ]);
}
```

Here we set `$tries = 0` so the worker never marks the job as failed. At the same time, we check if the job has been attempted more than 5 times, report the incident, skip the job (by returning `null`), and run the next job in the chain.

Minding the Payload Size

Laravel stores the chain's jobs payloads in the payload of the first job in that chain. That means if we chain too many jobs, the first job in the chain will have a massive payload, which can be a problem depending on our queue driver and system resources.

Instead of collecting all jobs from the `messages` database table, we can limit the results to only 10 messages on each run:

```
$messages = Messages::where(...)
            ->where(...)
            ->orderBy(...)
            ->limit(10)
            ->get();
```

Deactivating a Customer Account

Now we have a `Messager` job in queue for each customer. We want to find a way to remove this job from the queue in case we decide to deactivate a specific customer's account.

To do that, we can add a flag on the customer model `is_active = true,` and on each run of the `Messenger` job, we're going to check for this flag:

```
public function handle()
{
    if (! $this->customer->is_active) {
        return $this->fail(
            new \Exception('Customer account is not active!');
        );
    }

    $messages = //...

    //...
}
```

Now, if the `Messenger` job runs and finds the customer account deactivated, it will report the failure and not be retried again.

If the customer account was activated again, we would have to dispatch a new instance of the `Messenger` job to get the customer's messages processed.

Dispatching Event Listeners to Queue

So far, we've learned how to dispatch jobs to the queue. You can think of jobs as tasks triggered when certain events occur in an application. These tasks are dispatched to execute our business logic.

Laravel ships with an event dispatcher that listens to different events and executes several listeners we configure. Some of those listeners are better to be executed asynchronously.

Let's take a look at an application that's built using event-driven programming. When the user submits an order, a `NewOrderSubmitted` event is dispatched. This event has several listeners:

```
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        NewOrderSubmitted::class => [
            UpdateCustomerMetrics::class,
            SendInvoice::class,
            SendGiftCoupon::class,
        ],
    ];
}
```

The `NewOrderSubmitted` event is fired from a controller action:

```
class OrderController{
    public function store()
    {
        // ...

        event(new NewOrderSubmitted($order));

        return redirect('/thanks');
    }
}
```

Once the event is fired, Laravel will call the `handle()` method of all the listeners.

The `UpdateCustomerMetrics` listener updates the cache with the number of orders the customer has made, along with a few other metrics. We'll want this listener to run immediately so that when the customer views their dashboard, they'll see the correct numbers.

On the other hand, sending the invoice in the `SendInvoice` listener and rewarding the customer by sending a gift coupon in the `SendGiftCoupon` listener can be done later. We don't want the user to wait until those two emails are sent before they're redirected to the `/thanks` page.

Dispatching Listeners to The Queue

Here's how the `SendInvoice` listener looks:

```
use Illuminate\Contracts\Queue\ShouldQueue;
```

```
class SendInvoice implements ShouldQueue
{
    public function handle(NewOrderSubmitted $event)
    {

    }
}
```

By implementing the `ShouldQueue` interface on the listener classes, we're telling Laravel to send this listener to the queue instead of calling the `handle()` method immediately.

Inside a queued listener, we can configure the job attempts, backoff, timeout, retryUntil, delay, connection, and queue:

```
class SendInvoice implements ShouldQueue
{
    public $tries = 3;
    public $timeout = 40;
    public $backoff = [5, 20];
    public $delay = 10;
    public $connection = 'redis';
    public $queue = 'mail';

    public function retryUntil()
    {
        return now()->addHours(5);
    }
}
```

We can also define a `failed()` method which will be called if the listener job fails:

```
public function failed(NewOrderSubmitted $event, $e)
{
    //
}
```

Conditionally Sending Gift Coupons

For each new order, we must send an invoice email. But on the other hand, customers don't get rewarded with a gift coupon on each order. Instead, a loyalty points system is in place that determines if the customer is eligible for a reward after each purchase.

Here's how the `handle()` method of the `SendGiftCoupon` queued listener looks:

```
class SendGiftCoupon implements ShouldQueue
{
    public function handle(NewOrderSubmitted $event)
    {
        $customer = $event->customer;

        if ($customer->eligibleForRewards()) {
            $coupon = Coupon::createForCustomer($customer);

            Mail::to($customer)->send(
                new CouponGift($coupon)
            );
        }
    }
}
```

Each time the job runs, it will check if the customer is eligible for a reward.

Now, here's the real challenge; coupon eligibility can differ between the time the order was made and the time the job runs. As we explained in previous challenges, we have no control over when a worker will process a job; it might take a few milliseconds, and it might take several minutes.

In addition to this, customers are not going to be rewarded for each purchase. So dispatching the job to the queue and then deciding if a gift coupon should be awarded or not is a waste of resources.

Those jobs will reserve a spot in our queue while they're not going to do anything when they run.

Making the Decision Early

To solve the problems from above, we need to be able to make the decision right when the order is placed, not when the queued job runs. To do that, we're going to add a `shouldQueue` method to our listener:

```
class SendGiftCoupon implements ShouldQueue
{
    public function handle(NewOrderSubmitted $event)
    {
```

```
        $customer = $event->customer;

        $coupon = Coupon::createForCustomer($customer);

        Mail::to($customer)->send(
            new CouponGift($coupon)
        );
    }

    public function shouldQueue(NewOrderSubmitted $event)
    {
        return $event->customer->eligibleForRewards();
    }
}
```

When Laravel detects this method in a queued listener class, it will call it before sending the listener to the queue. If the method returns false, the listener will not be sent to the queue.

That way, we check the eligibility once the order is placed, which will make the decision more accurate and save valuable resources by not filling the queue with jobs that won't run.

Sending Notifications in the Queue

Similar to executing event listeners asynchronously, we can do the same with notifications. In this challenge, we want to send notifications to relevant users after each successful deployment. Some users prefer to receive a notification via email, and others prefer messaging platforms like Slack, Telegram, etc...

The `handle()` method of our deployment job may look like this:

```
public function handle()
{
    // Deploy Site...

    Notification::send(
        $this->site->developers,
        new DeploymentSucceededNotification($this->deployment)
    );
}
```

Depending on the size of the team and the notification preferences, sending all notifications

can take some time. Also, sending a notification on a specific channel may throw an exception if the channel service is down.

To avoid delaying other jobs in our deployments queue until notifications are sent after each deployment job, we're going to put that notification in a queue instead of sending them directly from inside the deployment job.

This will also help us prevent the scenario where sending a notification fails and throws an exception. In that case, our deployment job will be considered failed and might be retried while the main task -deploying a site- has already run successfully.

Queueying Notifications

Similar to queueing listeners, we can implement the `ShouldQueue` interface in the notification class to instruct Laravel to send it to the queue:

```
use Illuminate\Notifications\Notifictaion;
use Illuminate\Contracts\Queue\ShouldQueue;

class DeploymentSucceededNotification extends Notification
                                        implements ShouldQueue
{
    // ...
}
```

A separate job to send a notification will be queued for each notifiable and each channel. So if we send notifications to three developers on the Mail and Slack channels, six notification jobs will be sent to the queue.

If any notification sending jobs fail, they can be attempted multiple times or report failure like regular jobs.

We also configure how a notification is queued from inside the Notification class:

```
class DeploymentSucceededNotification extends Notification
                                        implements ShouldQueue
{
    public $tries = 3;
    public $timeout = 40;
    public $backoff = [5, 20];
    public $delay = 10;
    public $connection = 'redis';
```

```
    public $queue = 'mail';

    public function retryUntil()
    {
        return now()->addHours(5);
    }

    public function failed($e)
    {
        //
    }
}
```

We can also configure queue middleware to pass the jobs through before running:

```
public function middleware()
{
    return [new RateLimited];
}
```

Using Mailables

If we're only sending mail notifications, it might be more convenient to use Mailables instead of notifications. Queueing mailables works the same way as queueing notifications. You need to implement the `ShouldQueue` interface on the mailable class, and Laravel will send the email in a queued job.

You can also add public properties to the mailable class to configure tries, backoff, timeouts, and all the other queue configurations.

Dynamic Queue Consumption

In a previous challenge, we learned that we could dispatch jobs to multiple queues and configure how workers should prioritize those queues while starting the worker.

This worker, for example, will prioritize jobs from the `settlement` queue and will only consume jobs from the `default` queue when the `settlement` queue is empty:

```
php artisan queue:work --queue=settlement,default
```

However, in some cases, we may need to dynamically change the queue priority to achieve maximum utilization of our system resources.

In this challenge, we have a system that includes an electronic settlement engine. This engine transfers funds from an aggregation bank account to the accounts of several merchants. However, based on the commercial agreement, only 8000 transfers are allowed every hour.

To achieve this, you may consider putting a rate limiter on the settlement jobs, so they are released back to the queue when they hit the limit. However, with this approach, jobs for some merchants may get released several times and thus delay their settlement. We want to preserve the settlement order as much as we can. Unless there's a problem, merchants should receive their settlements based on a specific order.

It would be ideal to stop processing jobs from the `settlement` queue once we hit the limit and continue when the hourly window clears. To do that, we will use Laravel's *pop callbacks*.

Every time a worker is ready to process a new job, it calls a `pop` method on the queue driver implementation and specifies the queue name the popping should be from. Given that our worker consumes jobs from the `settlement` and `default` queues, you can see how the decision is made by checking figure 2-32.

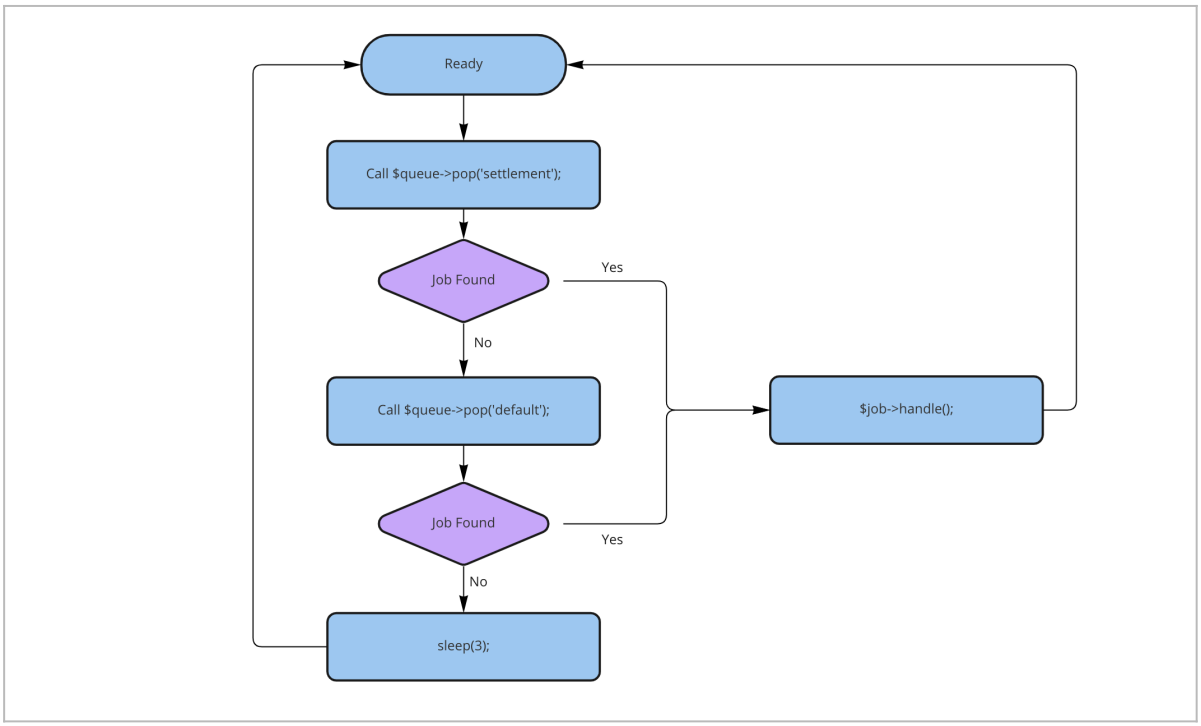

Figure 2-32. Poping jobs from different queues.

We can customize this behavior by defining a callback using the `popUsing()` static method on the `Worker` class. All we need is to give our workers a name and call the `popUsing()` method in the `boot()` method of one of the service providers.

To give a worker a name, you may use the `--name` option:

```
php artisan queue:work --name=settles-merchants --queue=default
```

> **Notice:** We can start as many workers as we want using the same name.

Now inside the `boot()` method of our `AppServiceProvider`, we're going to call `Worker::popUsing()` with `settles-merchants` as the first argument and a callback as the second argument:

```
namespace App\Providers;

use Illuminate\Queue\Worker;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        Worker::popUsing('settles-merchants', function($pop, $queue){
            //
        });
    }
}
```

The callback provided to `popUsing()` is invoked with two arguments:

1. The callback that pops a job from a specific queue.
2. The list of queues the worker is configured to consume jobs from.

To give you an example of how we can use this method, let's imagine we want to always pop jobs from the `settlements` queue regardless of what queues we provided to the worker command using the `--queue` option. Here's how we can do that:

```
Worker::popUsing('settles-merchants', function($pop, $queue){
    return $pop('settlement');
});
```

Now, whenever the worker is ready to pick a job up, it will pop one from the `settlement` queue, even if we don't have the `settlment` queue defined in the `--queue` option of the `queue:work` command.

Scheduling Consumption from the Queue

With the knowledge we have gained so far, we can store a counter in the cache for every time a worker processes a job from the `settlement` queue. When the counter hits the limit, we stop consuming jobs from that queue until the hour passes:

```
Worker::popUsing('settles-merchants', function ($pop, $queues) {
    $currentHour = now()->hour;
    $cacheKeys = 'settlement-'.$currentHour;
    $consumableQueues = explode(',', $queues);

    // If the counter value is less than the hourly limit, add the
    // 'settlement' queue to the list of consumable queues.
    if ((int) Cache::get($cacheKeys) < 8000) {
        array_unshift($consumableQueues, 'settlement');
    }

    foreach ($consumableQueues as $queue) {
        $job = $pop($queue);

        // If no job was popped, move to the next queue in the list.
        if (is_null($job)) {
            continue;
        }

        // If the job we just popped is from the 'settlement' queue,
        // we increment the counter.
        if ($queue == 'settlement') {
            Cache::increment($cacheKeys);
        }

        return $job;
    }
});
```

Here we construct the cache key based on the current hour, check if the counter for this

hour is within the limit and add the `settlement` queue to the beginning of the `$consumableQueues` array.

After that, we loop over all `$consumableQueues` and try to pop a job. We would check the next on the list if no job was found in a queue.

Finally, if a job was popped from the settlement queue, we increment the counter.

Figure 2-33 represents the decision-making process inside our `popUsing` callback. It guarantees we are prioritizing the `settlement` queue when we are within our settlement hourly limit. Otherwise, the worker will only consume jobs from the `default` queue, so it doesn't remain idle.

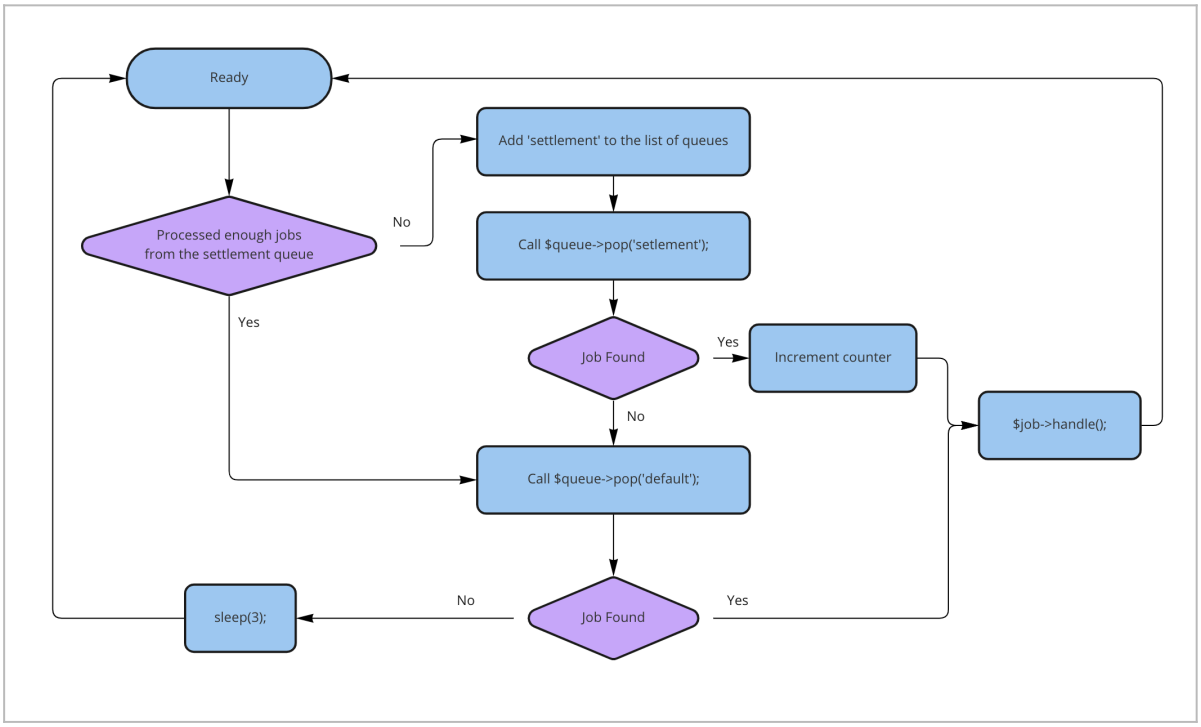

Figure 2-33. Poping jobs from different queues.

`popUsing()` is a very powerful tool in our toolbelt. It allows us to customize which queues we should consume jobs from at any time.

You should know that the callbacks are invoked before the timeout handler inside the worker is registered. That means your worker will get stuck if something is blocked during the callback execution. The timeout handler will not kill the worker and the process will stay in your server's memory doing nothing.

Customizing The Job Payload

We know by now that Laravel automatically converts our job objects, along with all configuration parameters, to a JSON string that can be stored in the queue driver.

The final array before the JSON encoding looks like this:

```
{
   "uuid":"67b31b39-26df-4b39-b2db-c4fb78088fd1",
   "displayName":"App\\Jobs\\ProcessPayment",
   "job":"Illuminate\\Queue\\CallQueuedHandler@call",
   "maxTries":null,
   "maxExceptions":null,
   "failOnTimeout":false,
   "backoff":null,
   "timeout":null,
   "retryUntil":null,
   "data":{
      "commandName":"App\\Jobs\\ProcessPayment",
      "command":"..."
   }
}
```

We also learned the importance of keeping our jobs self-contained. That we should provide all information a job may need to the job object, so it's available for the worker to process after deserialization. However, in some cases, we may not want to have to provide specific information to every single job. That's the case in our challenge.

In this challenge, we have a multi-tenant application where each tenant's data is isolated in a separate database. So, before processing every web request and every queued job, we need to extract the tenant ID to create a connection to the tenant database.

For web requests, the tenant ID can be stored in the session or token data. And since web requests are processed in a shared-nothing manner. The application will reset the state after every request and will throw an exception if the tenant ID is not included because it won't be able to connect to the tenant database.

However, for queued jobs, the story is different. Since a single application instance handles queued jobs, if we forget to connect to the database of the tenant the current job belongs to, the job may get processed on a database of a different tenant.

For that reason, we want to centralize the operation of storing the correct tenant ID for each job and connecting to that tenant's database when the job is picked up by a worker.

Using Payload Callbacks

Luckily, Laravel allows us to customize the payload of our jobs. And with that, I don't mean the parameters provided to the job class. I mean the entire array that Laravel decodes into JSON.

Using this, we can add a `tenant_id` attribute to the payload with each job dispatched. And to do so, we may use the `Queue::createPayloadUsing()` method inside the `boot()` method of one of our service providers:

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class TenancyProvider extends ServiceProvider
{
    public function boot()
    {
        $this->app['queue']->createPayloadUsing(
            function (string $connection, string $queue, array $payload) {
                return [
                    'tenant_id' => $this->app[Tenant::class]->id
                ];
            }
        );
    }
}
```

Here we use the container to resolve an instance of the queue manager, call the `createPayloadUsing()` method and provide a closure.

This closure returns an array that contains the tenant ID. In our example, we're extracting that ID from another container instance that resolves to the current tenants' object. The closure receives the name of the current queue connection, the name of the queue the job is being dispatched to, and the current job payload.

Laravel will merge whatever array we return from `createPayloadUsing` callbacks into the job payload while preparing the job for the queue. That means our job's final payload will look like this:

```
{
   "tenant_id": "f872339a-6587-4eec-a683-eddc22c3c9d5",
   "uuid":"67b31b39-26df-4b39-b2db-c4fb78088fd1",
   "displayName":"App\\Jobs\\ProcessPayment",
   "job":"Illuminate\\Queue\\CallQueuedHandler@call",
   "maxTries":null,
   "maxExceptions":null,
   "failOnTimeout":false,
   "backoff":null,
   "timeout":null,
   "retryUntil":null,
   "data":{
      "commandName":"App\\Jobs\\ProcessPayment",
      "command":"..."
   }
}
```

As you can see, the `tenant_id` is included in the payload and stored with all the other job attributes in the queue store.

Configuring The Tenant Object Before Job Processing

Now that we have the tenant ID stored in the payload, we can easily extract it before a job gets processed by listening to the `JobProcessing` event. The simplest way to do that is by registering a listener inside the `boot()` method of our service provider:

```
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessing;

class TenancyProvider extends ServiceProvider
{
    public function boot()
    {
        // ...

        $this->app['events']->listen(
            JobProcessing::class,
            function (JobProcessing $event) {
                $tenantId = $event->job->payload()['tenant_id'];

                // ...
```

```
            }
        );
    }
}
```

The job processing event is fired when a worker picks a job up from the queue before processing it. It includes an instance of `Illuminate\Queue\Jobs\Job`, which has a `payload()` method that we used to extract the `tenant_Id` attribute.

Now we can set the proper tenant object that will be used by our application while processing the current job. And if the `tenant_id` attribute doesn't exist in a job, an exception will be thrown, and the job won't get processed.

This defensive programming is essential when dealing with code in a shared-memory environment where the state is shared between operations. It helps us protect the system from very hard-to-catch bugs while the application is running in production.

CHAPTER 3

A Guide to Running and Managing Queues

Now that we've seen some real-life challenges and how we may deal with them using the queue system let's focus on understanding how the queue system works, how to manage it at scale, and how to monitor it.

We will also learn a series of practices to enable us to utilize our system resources efficiently. When using Laravel queues, it's fun to dive right into utilizing it without worrying about the price we pay. Unfortunately, system resources are neither unlimited nor free. Yes, some are cheap, but this can change when you operate at scale.

But before we talk tech, let's talk history. In late 2012, Taylor Otwell started working on a component that allows Laravel users to push jobs to one of two message queue services; IronMQ and Beanstalkd. It also included a console command, `queue:listen`, that keeps polling those services checking if there are any messages to read.

A job is a simple array that includes a class name and data to pass to that class. Laravel would take that array, serialize it using PHP's `serialize()` method, and send the string to the queue storage.

To push a job to the queue, you'd do something like this:

```
Queue::push('SendEmail', ['message' => $message]);
```

When the listener, now called *worker*, runs, it instantiates a `SendEmail` object and calls a `fire()` method with the data provided.

In 2013, Taylor would add the SQS driver, convert to JSON encoding instead of relying on PHP's serialization, introduce job timeouts and allow pushing jobs in bulk. Then, in September of that year, Taylor started experimenting with a Redis queue driver, which

would prove later to be the most advanced and flexible storage driver.

Throughout the years, and with the help of hundreds of contributors, Taylor took the queue component from a small experiment to a fully-fledged messaging system built into the framework. Available to all its users free of charge.

The idea is still as simple, though. An abstraction around sending an object to the queue store, and a console command that polls the store for jobs to process. It's put together in a way that is both very performant and very developer friendly.

Very rarely have people needed to understand how the queue system works under the hood. It just works. Still, I believe there will be times when this information becomes useful, and it's worth taking a moment to peek at how it works. And that's what we'll start this chapter with.

How Laravel Dispatches Jobs

There are two entry points to the queue system; one to dispatch jobs and another to process jobs. Let's start with the first entry point and examine what happens under the hood when a job is being pushed to the queue.

It all starts with constructing one of these three objects:

- `PendingDispatch`, when calling `Job::dispatch()`
- `PendingChain`, when calling `Bus::chain()`
- `PendingBatch`, when calling `Bus::batch()`

These objects prepare our jobs to be sent to the queue and expose a user-friendly API that we may use to configure how we want our jobs to be pushed to the queue and how we want them processed.

These objects are part of the *Bus component* that ships with Laravel. This component interacts with the queue component on our behalf. It's also responsible for preparing chains and batches and ensuring unique jobs are kept unique.

Figure 3-1 shows the bus and queue components' most important decisions to decide if/how a job should be dispatched to the queue. You can also see where the bus component ends, and the queue component starts.

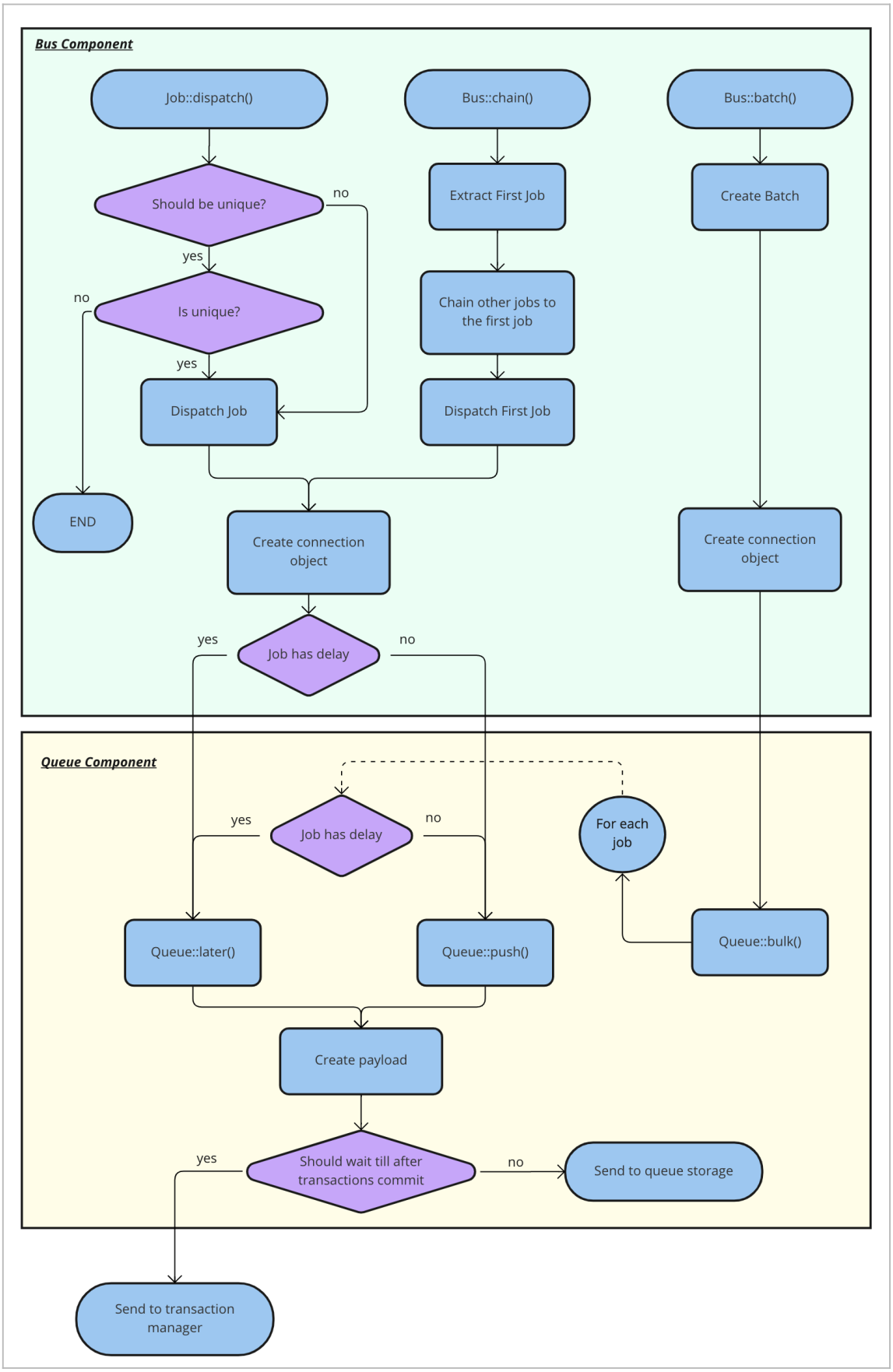

Figure 3-1. Poping jobs from different queues.

This diagram clearly shows that the unique jobs feature only works when dispatching single jobs to the queue. It doesn't work with batches or chains.

It's also clear that job chains work by adding all the jobs to the first job and dispatching it like a regular job. In other words, all the jobs are stored inside the first job's payload.

Deferring Dispatches Until Transactions Commit

In Figure 3-1, the queue sends the job to the transaction manager if it's configured to be dispatched after transactions commit. This is a critical decision as it helps us avoid a race condition when the job runs before the transaction commits, which leads to failures because the job reads a version of the database that doesn't include any changes applied within the transaction.

Laravel does that by wrapping the code that pushes the job to the queue inside a callback. When we attempt to push a job to the queue while we've configured the queue to check for active transactions, Laravel checks if there are active transactions and stores this callback in an internal collection inside the `DatabaseTransactionsManager` component. Otherwise, it executes the callback, pushing the job to the queue.

Figure 3-2 shows a logical view of how this works. When the transaction manager detects an active transaction, it stores the callback inside a collection that groups items by transaction levels. Level 1 represents any work captured right after a transaction begins. All subsequent levels represent additional savepoints made within the transaction.

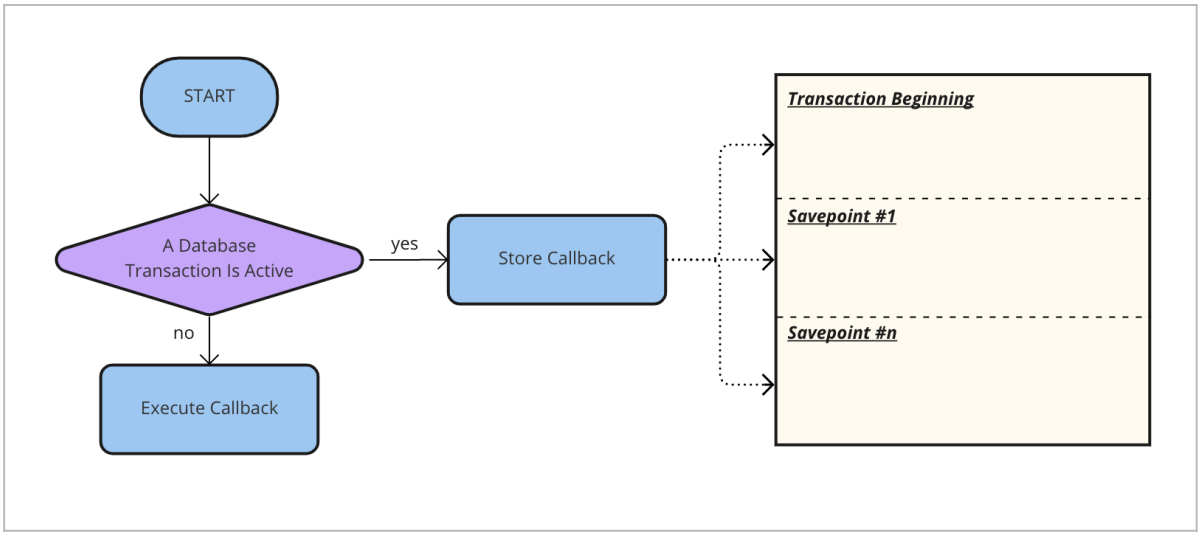

Figure 3-2. Storing the pushing callback in the transaction manager.

Laravel abstracts the queries to start, commit, roll back transactions, and savepoints. So here's a code snippet that shows what each level represents in Laravel code:

```
DB::transaction(function(){
    // Level 1

    DB::transaction(function(){
        // Level 2 (savepoint 1)

        DB::transaction(function(){
            // Level 3 (savepoint 2)
        });

        // Level 3 (savepoint 2)
    });

    // Level 3 (savepoint 2)

    DB::transaction(function(){
        // Level 4 (savepoint 3)
    });

    // Level 4 (savepoint 3)
});
```

Notice that whenever a new level starts, all work done is stored under this level. Even work outside the `DB::transaction()` call that started the level. You can see that better in figure 3-3.

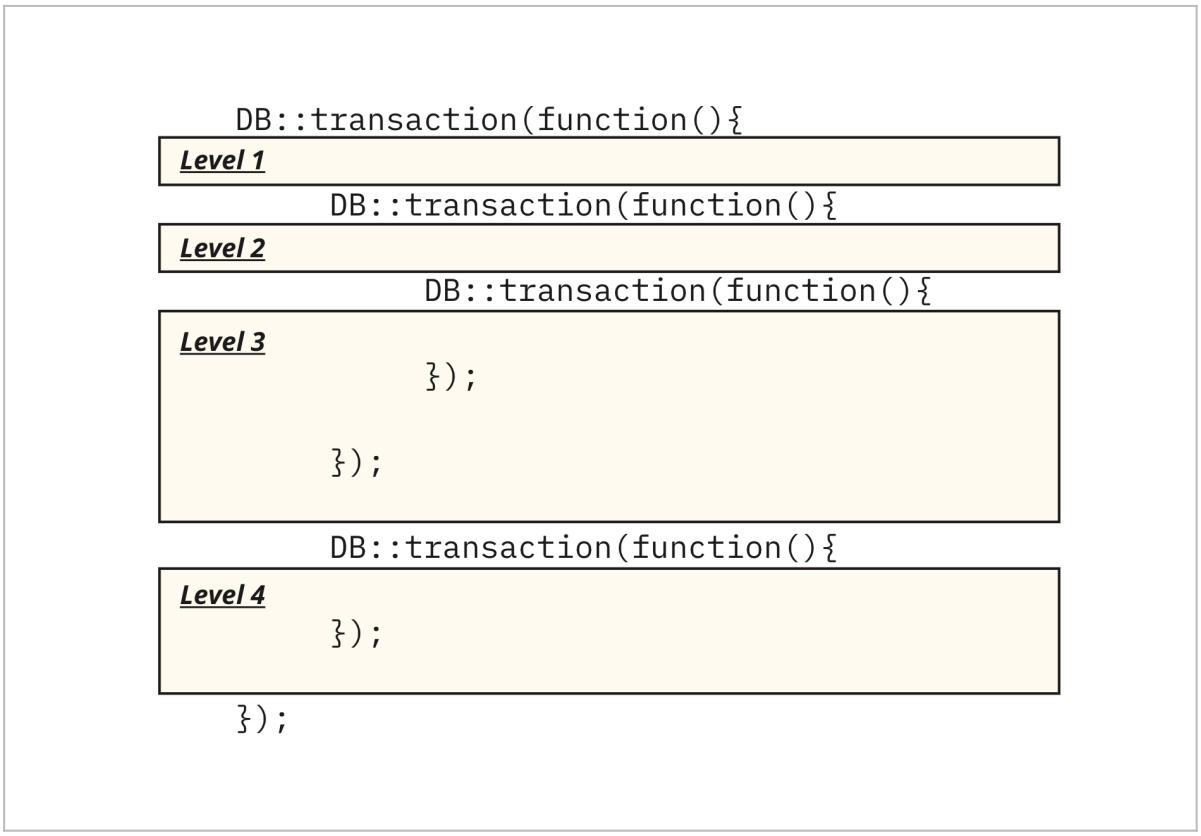

Figure 3-3. The beginning and end of each transaction level.

With deferred callbacks stored for each level, Laravel waits until the primary transaction commits and executes all registered deferred callbacks in all levels.

If any exception was thrown in any level, Laravel will roll back the entire transaction and deletes all deferred callbacks. That means a job dispatched within a transaction that was rolled back will not make it to the queue.

Also, if the user explicitly rolled back to a specific level, Laravel will delete all deferred callbacks registered after this level (database savepoint).

To conclude this part, the main benefits of sending job dispatches to the transaction manager component are:

1. Jobs won't get dispatched until transactions commit.
2. Jobs won't get dispatched if transactions are rolled back.

Dispatching Listeners, Mail, Notifications, and Broadcasted Events

Laravel abstracts the work needed to execute many common tasks asynchronously by sending them to the queue. Take event listeners, for example; instead of having to dispatch a job from a listener to do some work asynchronously, Laravel allows you to mark the

listener as `ShouldQueue`, and it will take care of sending it to the queue for you.

`ShouldQueue` is an interface that, when added to a listener, mail, notification, or broadcasted event, instructs Laravel to execute that task in the queue.

When Laravel detects it in a listener, it dispatches a `CallQueuedListener` job instead of running the listener `handle()` method. This `CallQueuedListener` job receives the listener class name and any event data in the constructor. It uses this information to construct the listener when the job runs and executes it.

While constructing this `CallQueuedListener` job, the event dispatcher copies all the queue properties we may have on the listener to the job. So, for example, if you have a `$tries` public property on the listener, the dispatcher copies it to the `CallQueuedListener` job. This allows you to configure the queued listener like you'd configure a job.

The same principle is applied to mailables, notifications, and broadcasted events. Laravel detects the `ShouldQueue` interface and pushes a job to the queue instead of executing the task.

One thing you need to know is that this job is sent to the queue component directly without passing through the bus component. That means we cannot use unique jobs, chains, or batches when sending those tasks to the queue.

Dispatching a Closure

In addition to dispatching objects, Laravel allows us to dispatch closures as well:

```
dispatch(function () use ($report) {
    $report->generate();
});
```

Under the hood, Laravel serializes the closure using its own `SerializableClosure` component, which converts any closure to a simple object that can be sent over the network and stored inside any data store.

After serialization, the serializable object is used to construct a `CallQueuedClosure` object. This object is the job that is dispatched to the queue. When the job runs, it will unserialize the closure and invoke it.

The `SerializableClosure` component is used to serialize any closure, not just queued

closures. This includes the closures passed to the `catch()`, `then()` and `finally()` methods when dispatching a batch, the `catch()` method when dispatching a chain, and also closure-based routes.

The payload of the closure we just dispatched will look like this:


```
{
   "uuid":"04818915-91d8-41c9-aac2-de6a624f1085",
   "displayName":"Closure (web.php:21)",
   "job":"Illuminate\\Queue\\CallQueuedHandler@call",
   "maxTries":null,
   "maxExceptions":null,
   "failOnTimeout":false,
   "backoff":null,
   "timeout":null,
   "retryUntil":null,
   "data":{
      "commandName":"Illuminate\\Queue\\CallQueuedClosure",
"command":"O:34:\"Illuminate\\Queue\\CallQueuedClosure\":1:{s:7:\"closure\";O:47:\"Larav
el\\SerializableClosure\\SerializableClosure\":1:{s:12:\"serializable\";O:46:\"Laravel\\
SerializableClosure\\Serializers\\Signed\":2:{s:12:\"serializable\";s:444:\"O:46:\"Larav
el\\SerializableClosure\\Serializers\\Native\":5:{s:3:\"use\";a:1:{s:4:\"report\";O:45:\
"Illuminate\\Contracts\\Database\\ModelIdentifier\":4:{s:5:\"class\";s:15:\"App\\Models\
\Report\";s:2:\"id\";i:1;s:9:\"relations\";a:0:{}s:10:\"connection\";s:5:\"mysql\";}}s:8
:\"function\";s:58:\"function () use ($report) {\n          $report->generate();\n
}\";s:5:\"scope\";s:37:\"Illuminate\\Routing\\RouteFileRegistrar\";s:4:\"this\";N;s:4:\"
self\";s:32:\"0000000000000132000000000000000\";}\";s:4:\"hash\";s:44:\"h8tP9k2tQfl00f3
ZkATg6L0ufDDowoeSktDkpfVWT1Y=\";}}}"
   }
}
```


If you take a deeper look at the `data.command` attribute, you will be able to see the code of the closure stored as a string:

```
function () use ($report) {\n        $report->generate();\n    }\
```

To prevent the security risk of someone altering the code before the job is processed, Laravel signs the string and includes the hash in the payload. Then, while the job is being unserialized, Laravel verifies the signature and throws an `InvalidSignatureException` exception if it doesn't match.

Creation of The Job Payload

In Figure 3-1, the final task performed by the queue component before sending the job to the queue store or the transaction manager is creating the job payload. This process passes through five stages:

1. Extracting and assigning the configuration attributes.
2. Serializing the job class.
3. Encrypting the serialized job (when needed).
4. Invoking payload hooks.
5. Encoding the payload array into JSON.

Extracting the Configuration Attributes.

Let's take a look at a snippet from Laravel's core:

```
[
    'uuid' => (string) Str::uuid(),
    'displayName' => $this->getDisplayName($job),
    'job' => 'Illuminate\Queue\CallQueuedHandler@call',
    'maxTries' => $job->tries ?? null,
    'maxExceptions' => $job->maxExceptions ?? null,
    'failOnTimeout' => $job->failOnTimeout ?? false,
    'backoff' => $this->getJobBackoff($job),
    'timeout' => $job->timeout ?? null,
    'retryUntil' => $this->getJobExpiration($job),
];
```

Most of the values of these configuration attributes are extracted from the job class, either from its properties or methods, except for the `uuid` and `job` attributes.

The `uuid` attribute is a unique identifier for the job generated when dispatching it. The `job` attribute is a hardcoded string that references the `call()` method of the `CallQueuedHandler` class. When workers pop the job from the queue, this is the method it will call. We'll learn more about that when we discuss how Laravel processes jobs from the queue.

The `CallQueuedHandler@call` method accepts the value of the `data` attribute from the payload. This attribute holds an array with two elements; the `commandName`, which contains the name of our actual job class, and the `command`, which has the serialized, possibly

encrypted, version of the job object.

```
[
    'data' => [
        'commandName' => get_class($job),
        'command' => serialize(clone $job),
    ]
];
```

Serialization of the Job Object

When you serialize a PHP object, its state is converted into a string representation. This string can be stored in any data store and can be used to reconstruct a copy of that object.

Laravel uses PHP's built-in `serialize()` function to serialize job objects. However, as you can see from the code above, Laravel first clones the job object before passing it to the serialize function. Let's explore why.

While serializing a job, we may need to transform some object properties. For example, we learned that Laravel transforms any eloquent model instances to a `ModelIdentifier` before serialization. That means if we try to access a model instance after dispatching a job, we will get a `ModelIdentifier` instead.

```
$invoice = Order::find();

$job = SendInvoice::dispatch($order);

get_class($job->order); // Illuminate\Contracts\Database\ModelIdentifier
```

Now, you may think that accessing job properties after dispatch is not a common use case. And you're right. However, transformation is not only needed for the properties of the job object itself. It might be needed for properties of some of the dependencies as well. Let's take a look:

```
class SendInvoice
{
    public function __construct(Invoice $invoice)
    {
        $this->invoice = $invoice;
    }
}
```

```
class Invoice
{
    public function __construct($pdf)
    {
        $this->pdf = $pdf;
    }
}
```

In this example, the `SendInvoice` job has a dependency on an object of class `Invoice`. This object has a `pdf` property which holds a file. Since PHP doesn't play well with serializing files, we need to convert the file to its path while serializing. And then bring the file instance back using its path when we unserialize.

```
class Invoice
{
    public function __construct($pdf)
    {
        $this->pdf = $pdf;
    }

    public function __serialize()
    {
        return ['pdf' => $this->pdf->getFilename()];
    }

    public function __unserialize(array $data)
    {
        $this->pdf = new SplFileInfo($data['pdf']);
    }
}
```

Using the `__serialize()` magic method, we returned a serialization-friendly representation of the object. Then with `__unserialize()`, we restored the object's properties from that representation.

As you may have noticed, using `__serialize()`, we didn't have to do any transformation to the object properties. The problem happens for objects that use `__sleep()` instead:

```
class Invoice
{
    public function __construct($pdf)
    {
        $this->pdf = $pdf;
```

```
    }

    public function __sleep()
    {
        $this->pdf = $this->pdf->getFilename();

        return ['pdf'];
    }

    public function __wakeup()
    {
        $this->pdf = new SplFileInfo($this->pdf);
    }
}
```

When using `__sleep()`, we only return the keys that should be included in the serialized representation, unlike `__serialize()`, where we can return the serializable value with the key. That means we have to transform the value of the actual property when using `__sleep()`. This means, we'll affect the `Invoice` object beyond dispatching the job:

```
$invoice = new Invoice(/****/);

$job = SendInvoice::dispatch($invoice);

dump($invoice->pdf); // This prints the filename, not the file.
```

Since `__serialize()` and `__unserialize()` are available as of PHP `7.4.0`, many applications still use `__sleep()` and `__wakeup()`. To handle these applications, Laravel has to clone the job object so that transformations happen on the clone, not the object itself.

Now, these applications can use the `__clone` magic method to instruct Laravel to clone the job object dependencies when the job object is being cloned:

```
class SendInvoice
{
    public function __construct(Invoice $invoice)
    {
        $this->invoice = $invoice;
    }

    public function __clone()
    {
        $this->invoice = clone $this->invoice;
```

```
    }
}
```

When Laravel clones this job, the invoice property of the job will hold a clone to the original invoice object, any transformations inside the invoice's `__sleep()` method will only apply to the clone, not the original object.

Encrypting the Job Object

Now that we know the `data.command` attribute of the job payload holds the serialized version of the job object, we may expect a job payload to look like this:

```
{
   "data":{
      "commandName":"App\\Jobs\\VerifyUser",
"command":"O:19:\"App\\Jobs\\VerifyUser\":1:{s:41:\"\u0000App\\Jobs\\VerifyUser\u0000soc
ialSecurityNumber\";s:9:\"45678AB90\";}"
   }
}
```

This is a `VerifyUser` job with a property called `socialSecurityNumber`. If you look closer, you can see the social security number stored as a string (`45678AB90`) visible to the human eye. Any person—or program—that gains access to the queue store will be able to extract this value.

For most jobs, this isn't a problem. But if the job stores critical information in the payload, it's better we encrypt it so that only our queue workers can read it while processing the job.

To do that, we can utilize the `__serialize()` and `__unserialize()` PHP magic methods to encrypt and decrypt the social security number while serializing and unserializing. However, Laravel allows us to do the encryption for us by simply implementing the `ShouldBeEncrypted` interface on the job class.

When we implement this interface, Laravel will encrypt the entire `data.command` attribute for us before storing the job in the queue. It will also decrypt it when workers pick the job up.

Here's how the payload may look when we implement `ShouldBeEncrypted`:

```
{
   "data":{
```

```
      "commandName":"App\\Jobs\\VerifyUser",
"command":"eyJpdiI6InBWNVIwTXFJOGwwNXE2QzI5SFRZWlE9PSIsInZhbHVlIjoiQzFqZUZkTDBVcmk3SjJoS
1VtZ3llTkZWTDdPOHVVSUhJTVdpOE15WFcyU1VUN2dFaHdQOE5ZZHB2WTdVekxwVGJ4Slhqc2VNMTlCcmM3TkpCb
2JjZXI2c0ZEZHNkSGtDSkVNN3ZrcHlHZ2g5bHNJaGZWdktYTDR5MzVoc0R4MUh6SlZmS2M3OTYweTFCOVQ1RUxUY
UhRPT0iLCJtYWMiOiJkZWNjYmNiNmYwN2Y2NGNkNDc2NWE0YTJmYWVhYzVmODM4Y2U3YmUxMTZhYTM0OTk3ZmY5Y
2Y4MTY0MzYxY2E5IiwidGFnIjoiIn0="
   }
}
```

Now, all sensitive information stored in the job object will not be available to digest.

Invoking the Payload Hooks

Laravel allows us to customize the payload of our jobs by calling a `Queue::createPayloadUsing()` method into one of our service providers. We have seen this in action in one of the challenges in this book.

While Laravel is preparing the payload, it will call all payload callbacks and merge the results into the generated job payload. Here's a snippet from the Laravel core:

```
foreach (static::$createPayloadCallbacks as $callback) {
    $payload = array_merge(
        $payload,
        $callback($this->getConnectionName(), $queue, $payload)
    );
}
```

Encoding the Payload into JSON

Before sending the job to the queue, the final stage is encoding the PHP array into JSON. If that step fails, an `InvalidPayloadException` will be thrown.

How Workers Work

With the jobs persisted in one of the queue storage engines (Redis, Database, SQS, etc...), the queue system finishes one of its two tasks. The other task, as you can imagine, is processing these jobs. And from the knowledge we gained from the previous chapters, we know that it all starts with running a console command:

```
php artisan queue:work
```

This command starts a PHP process that loads the Laravel application in the server memory and bootstraps it. This is similar to every artisan command that ships with Laravel and artisan commands that we would add to our applications. What's unique about the `queue:work` command is one of the dependencies it requires in its constructor, which is an `Illuminate\Queue\Worker` instance.

The `Worker` class is the engine of the queue component. And to show you how it works, let me compare it to internal combustion engines. These engines produce power by relying on a repeating pattern of air & gas intake, compression, power through combustion, and exhaust.

Similarly, the worker engine relies on a repeating pattern of poping, preparing, processing, and cleaning. This infinite loop is powered by PHP's `while` control structure:

```
while (true) {
    $job = pop();

    prepare($job);

    process($job);

    clean();
}
```

Having `while (true)` here means this loop will run endlessly as the condition for the loop will remain `true`, which is basically what we need. A block of code is repeated repeatedly (Figure 3-4).

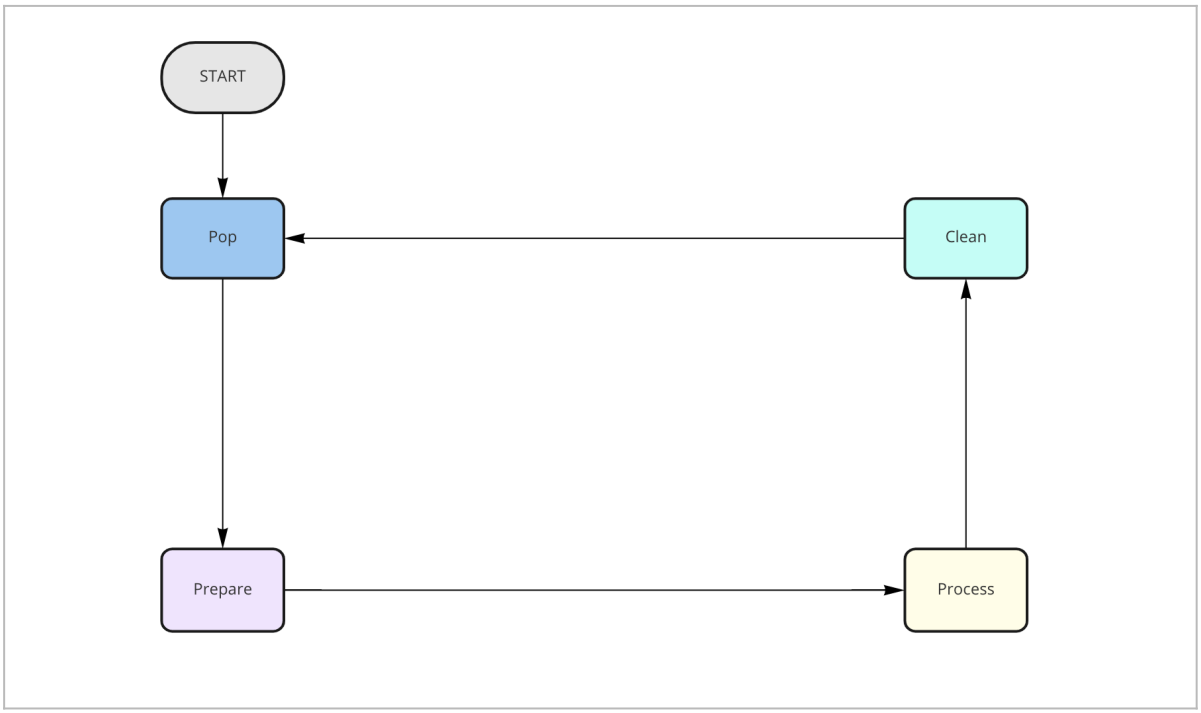

Figure 3-4. The worker loop.

The cycle isn't as simple as it seems, though. Each of these phases executes several procedures to ensure the engine is working in a reliable and efficient manner. So let's expand on the previous figure to show some of these procedures:

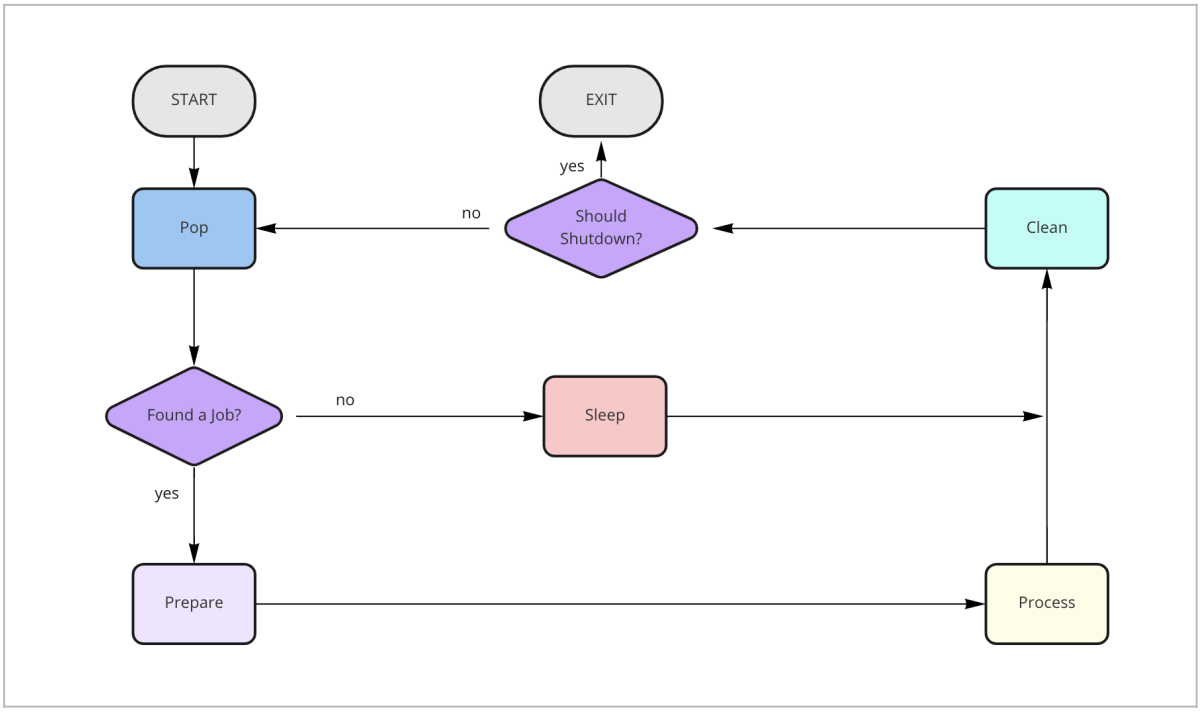

Figure 3-5. The worker loop.

The first thing I want you to observe in figure 3-5 is that if the worker couldn't find a job to process—the queue is empty— it's going to sleep for a few seconds, skip the preparing and

processing phases and jump to the cleaning phase.

The reasons for sleeping when the queue is empty are:

1. Avoid bombarding the queue storage engine with too many dequeue commands while this queue is empty.
2. Yield the CPU time so other programs can use it to do meaningful work.

By default, Laravel sleeps 3 seconds when it can't find a job to process, trading immediacy for the reduced load. This means if a job becomes available just ten milliseconds after the worker goes to sleep, it will wait in the queue for almost three full seconds before the worker wakes up and pops it.

You may reduce the amount of time the worker sleeps by configuring the `--sleep` option on the `queue:work` command:

```
php artisan queue:work --sleep=0.5
```

The second thing to notice in figure 3-5 is that the worker checks if it should shut down before starting a new cycle. This allows us to break the loop and end the program when needed.

We know that there's always the option to kill a process that enters an infinite loop immediately using the `KILL` command and sending a `SIGKILL` signal:

```
kill -9 [PROCESS_ID]
```

However, doing so while the worker is in the middle of processing a job could lead to data corruption. So, Laravel recommends killing the process using a `SIGTERM` signal instead. Laravel will catch this signal and set a `shouldQuit` property to `true`. Before the worker starts a new loop, it will check this property—among other things—to decide if it should shut down. This is called graceful termination.

Signal Handling

Speaking of signals, workers listen to two kinds of signals; operating system signals and cache signals. It uses these signals to decide if a worker should shut down and what work it should do before it does.

When the worker starts, before entering the first loop, it registers some OS signal handlers

for handling termination and pause signals. It also reads from the cache and stores the timestamp of the last restart signal (Figure 3-6).

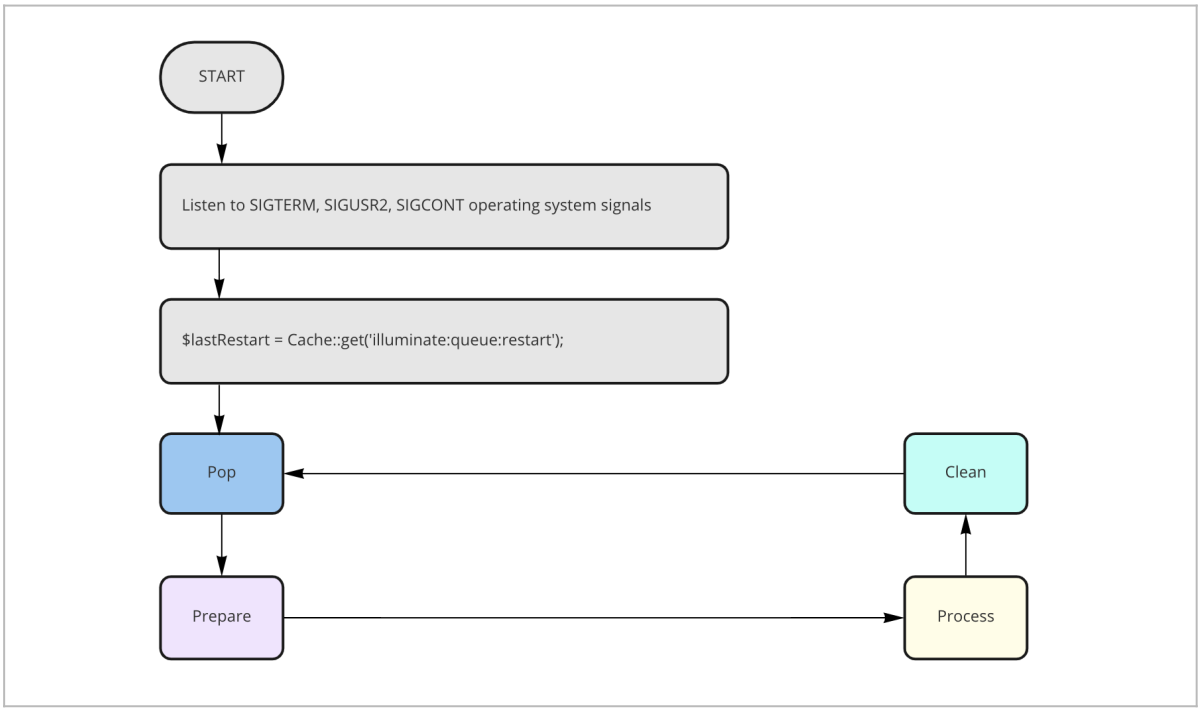

Figure 3-6. Listening for signals.

Inside the OS signal handlers, Laravel listens to a `SIGTERM` signal and sets the `shouldQuit` property to true. It also listens to a `SIGUSR2` signal and sets the `paused` property to true. And finally, it listens to a `SIGCONT` signal and sets the `paused` property to false.

Now during the cleaning phase of the worker cycle, Laravel will check if `shouldQuit` is set to `true` and shutdown the worker. It'll also read a fresh value of the `illuminate:queue:restart` cache key and shut down the worker if the timestamp does not match the one stored when the worker started. Because if the timestamp did not match, it means a recent restart signal was sent after the worker started, so it needs to shut down so it's started again.

> **Notice:** You can instruct all your workers to restart by calling the `queue:restart` command. This command will store the current timestamp in the `illuminate:queue:restart` cache key.

Also, during the cleaning phase, the worker will check if `paused` is set to `true` and pause for a while by repeatedly calling the `sleep()` PHP function until `paused` is set to `false`. Having this in place means you can instruct a worker to pause processing any jobs until you enable it back again.

The remaining signal the worker handles is `SIGALRM`, which is an automatic signal sent by the PHP process control extension when the internal alarm goes off. The timer for the alarm starts during the popping phase and is disabled during the cleaning phase.

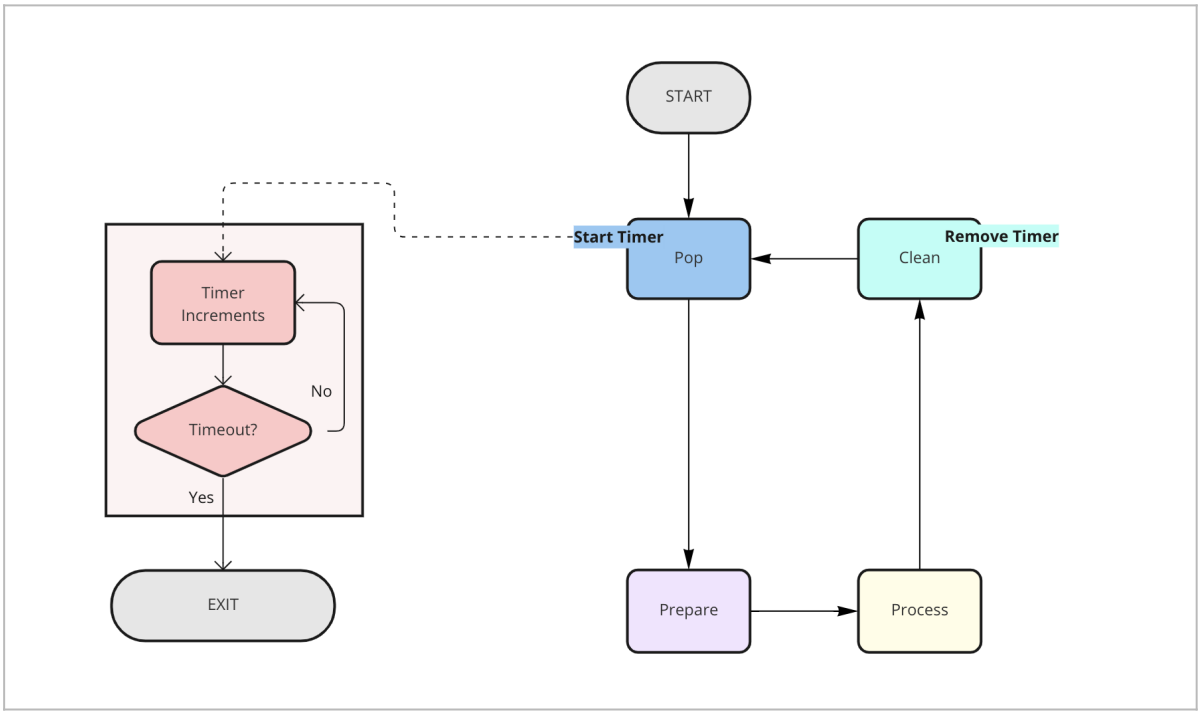

Figure 3-7. The timeout alarm handler.

> **Notice:** We already mentioned earlier that you could set the timeout on the worker level using the `--timeout` console option or on the job level by setting the `$timeout` job class property.

When the `SIGALRM` signal is caught, the worker will exit immediately. Even if there's a job being processed. But before it exits, it'll check if the job should be marked as failed depending on its number of tries and maximum allowed exceptions.

Preparing And Processing The Job

When a job is popped from the queue, the worker prepares it to be processed. This start with checking if the job has exceeded its maximum attempts already and marking it as failed in that case.

Next, Laravel will execute the `call` method of the `CallQueuedHandler` class and pass the `data` attribute of the payload. Inside this method, the job instance is decrypted, unserialized, passed through the middleware, and the `handle()`—or `__invoke()`—method

is executed.

While unserializing the job, Laravel catches any `ModelNotFoundException` exceptions, sends the job to the dead-letter queue, and deletes it from the queue. Or, if a `deleteWhenMissingModels` job property is set to `true`, it will delete the job immediately without moving it to the dead-letter queue.

Finally, suppose the job is configured to be unique until it's picked up by a worker. In that case, Laravel is going to release the unique lock—before passing the job through the middleware—so that other instances of the job can be dispatched to the queue.

> **Notice:** You can configure a job to be unique until processing by implementing the `ShouldBeUniqueUntilProcessing` interface on the job class.

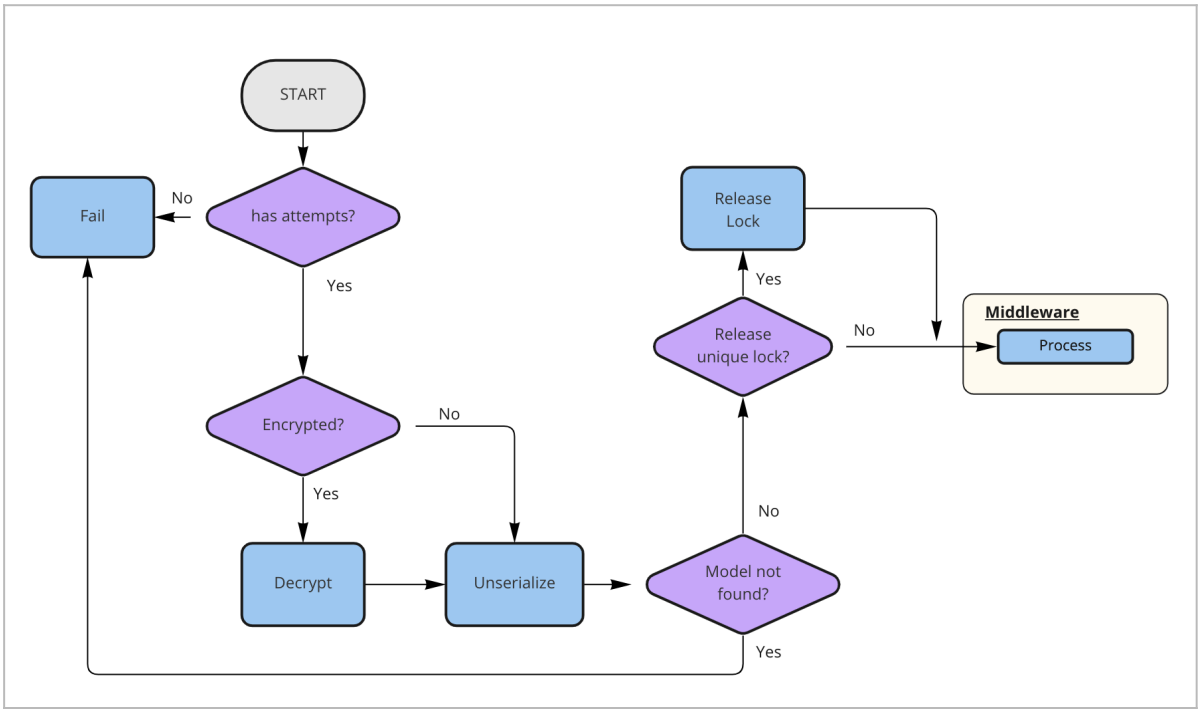

Figure 3-8. Preparing and processing the job.

The Cleaning Phase

After processing the job, the worker will check if any exception was thrown during processing. In that case, it'll inspect the exception and decide on three things:

1. If a failed database connection caused the exception, the worker will mark `causedByLostConnection = true` so that it shuts down after finishing the cleanup.
2. If the job has exhausted all allowed attempts or exceptions, the worker will move it to

the dead-letter queue and delete it from the queue.
3. If the job hasn't exhausted all attempts, the worker will release it back to the queue to be retried later.

In case the job has exhausted all attempts, the worker will also do the following:

1. Release any unique locks.
2. Invoke all `catch()` callbacks of the chain if the job is part of a chain.
3. Update the batch information and invoke the `catch()` callbacks if the job is part of a batch.

On the other hand, if the job is completed successfully, the worker will do the following:

1. Release any unique locks.
2. Run the next job in the chain, if there's any.
3. Update the batch information if the job is part of a batch.

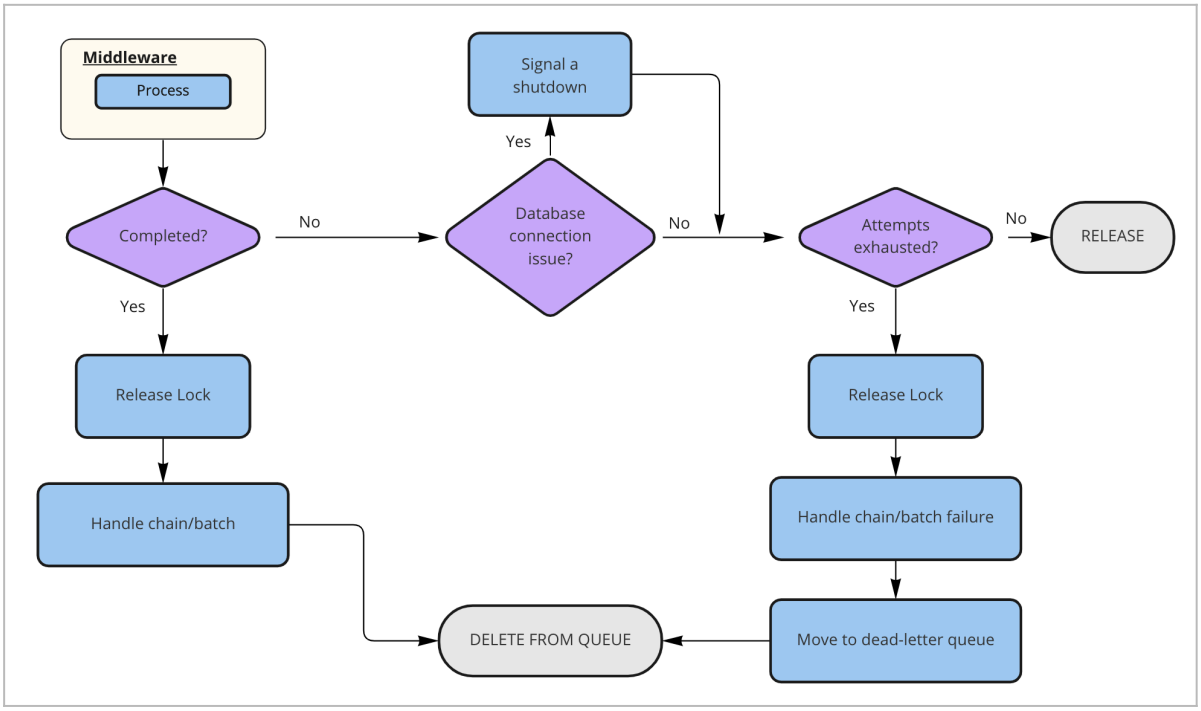

Figure 3-9. Cleaning after a job.

Next step, the worker will check if it should shut down or pause. A worker may shutdown if:

1. `shouldQuit` was set to `true`.
2. The maximum memory was exceeded.
3. The maximum number of jobs processed was reached.
4. The maximum lifetime of the worker was reached.
5. A restart signal was sent.

And it may pause if `paused` was set to `true`, or the application is in maintenance mode.

We can enforce the worker to continue processing jobs while the application is in maintenance mode by setting the `--force` option in the `queue:work` command.

The final step in the cleaning phase is removing all scoped instances from the Laravel container. These instances are singletons that should only live to process a single request or a single job. The worker removes these instances, so fresh ones may be created when the new job needs them.

Choosing the Right Machine Configuration

Now that we know how the queue system works let's investigate how the worker process utilizes system resources. Understanding this will help us choose the right machine type for our queue workloads.

Most cloud providers nowadays offer a wide variety of machine types; some are optimized for compute-intensive workloads, while others are optimized for memory-intensive workloads.

Finding the ideal machine type for every use case is a long-term exercise that requires several iterations. Under-resourced machines lead to jammed queues, long delays, timeouts, and potential crashes. On the other hand, over-resourced machines will cost us money for nothing.

In this chapter, I want to give you a general idea of what to look for, in terms of CPU and memory, when configuring choosing a machine to run your queue workloads.

Let's start with a simple setup, a machine with one dedicated vCPU backed by a processor that runs 2.7 GHz. That processor can produce 2,700,000,000 cycles per second. Each instruction it executes may take one or more cycles to complete. Instructions can vary between storing a variable in memory and compressing a large video file.

Let's start one worker on this machine:

```
php artisan queue:work
```

This worker process will use the CPU cycles available to execute the instructions needed to pop a job from the queue and execute it. If no job was found, the loop would continue searching over and over. Along the way, it will also do some housekeeping, as explained in

the previous chapter.

Even though this process is an infinite loop, it won't be using all CPU cycles all the time. Some of the time will be spent waiting for a response from other services. The kind of work that requires waiting is called I/O bound. A few examples of this are:

1. Using the Redis queue driver, the worker waits for a response from the Redis instance each time it tries to fetch a new job.
2. When reading the fresh timestamp of the restart cache key.
3. All communication with the database while processing a job.
4. All communication with external HTTP services while processing a job.
5. Reading and storing files on disk.
6. Sleeping.

While waiting for I/O tasks, the processor cycles can be utilized by other processes on the machine. Figure 3-10 shows our process popping and processing jobs. The gaps between rectangles represent waiting for I\O bound tasks.

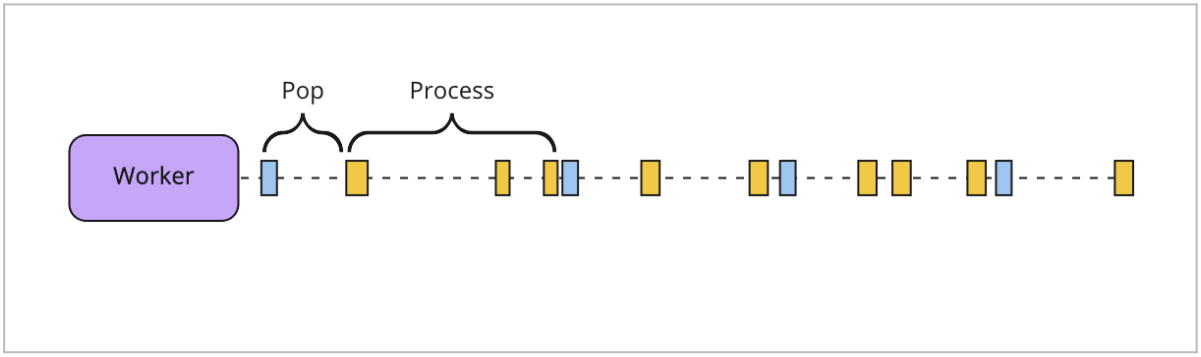

Figure 3-10. CPU bound vs. I/O bound work.

Light Queue

Let's think about the condition where the queue is empty. In that case, the worker will keep repeatedly communicating with the queue store to pop jobs. Continuous, non-stop communication will keep the CPU busy, even though the queue is empty.

Since there are no jobs to process, I\O bound work will be limited to the popping phase and some housekeeping. This means the CPU will be utilized more (Figure 3-11).

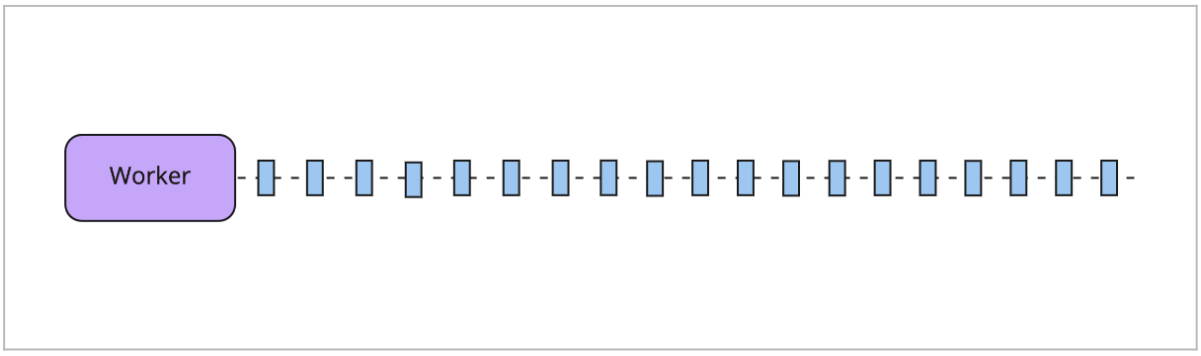

Figure 3-11. Empty queues lead to CPU greedy workers.

Keeping the CPU busy with all this popping that returns empty-handed may be considered a waste of resources for a machine that has just one vCPU like this one. This CPU time can be allocated to do other meaningful work, like handling HTTP requests or running CRON jobs. That's why Laravel sleeps for a while if it finds the queue empty. To yield the CPU time so other processes can use it.

Busy Queue

Now let's assume that our queues will always be full; a worker will always find jobs to run. So the number of CPU cycles a worker will utilize depends on the kind of jobs it runs.

If the jobs mostly execute CPU-intensive operations, that worker will utilize all the CPU cycles it can steal. An example of such a job is one that loops over millions of items in an array to calculate some metrics. This process will only yield at the end of the last iteration. While looping, the process will be in constant need of CPU time. This makes it an expensive process.

```
$totalDuration = 0;

foreach ($logs as $log){
    $totalDuration += $log['session_duration'];
}
```

On the other hand, if the jobs involve a lot of waiting, then the worker process is going to spend most of its time idle. As a result, CPU time will be available for other processes to utilize.

Deciding on The Number of Workers to Run

Now that you know the difference between the CPU-bound and I\O-bound work a worker process may perform. There are two more things you need to know before deciding on the

number of workers this on-vCPU machine can run.

The first thing is that most of the work we do in our Laravel applications is I\O pound work; running database queries, sending mail, running cache commands, sending requests to APIs, reading and storing files, etc...

The second thing is that modern operating systems have a process scheduler that determines which processes should utilize the CPU time at any given moment. Its goal is to keep the CPU utilized at all times and finish running all programs as soon as possible. This process scheduler can decide that a specific process has enough CPU share, put it in a READY state (meaning that it's paused), and switch the CPU to work on another process.

This sort of forced yielding is called *Preemptive Scheduling*, which is why a single vCPU can run multiple processes concurrently even if these processes don't yield much.

With a workload mostly I\O bound and a scheduler that switches CPU-intensive ones, we can run multiple Laravel workers on a machine with a single vCPU. Worker processes, application processes, and OS processes will share the CPU time to run tasks concurrently.

The limiting factor here will be the average waiting time (or AWT), which is the average time processes spend waiting for the scheduler to pick them for execution. The higher AWT, the slower your server seems.

Too many processes will lead to 100% CPU utilization, which is not a bad thing. However, we are paying for this CPU whether we are utilizing it fully or not. The problem is a noticeably high AWT, not 100% CPU.

A high AWT means you won't be able to run quick commands on the machine, even SSHing and navigating through directories will seem slow. Moreover, if the server handles web requests, some of these requests will timeout because the machine takes too long to respond. And of course, some jobs will timeout and get released back to the queue.

Scaling the server to have more vCPUs will allow us to run more workers without increasing the AWT. In this scenario, multiple CPUs will work in parallel to perform our compute tasks.

Also, this true parallelism means our queue will be processed faster because multiple workers will be working on processing jobs simultaneously. Not concurrently.

Using a Shared vCPU

In a machine with shared vCPUs, our ability to utilize CPU cycles depends on the workload of the neighboring machines. Therefore, if we share a vCPU with machines under heavy load,

we will not be able to utilize the full power of the CPU. Moreover, the server provider may send us a warning if we push hard for too long.

Shared vCPUs are an excellent option for low to medium workloads with occasional bursts for brief periods. They are cheaper than machines that come with dedicated vCPUs, so I recommend you start with this option if you're not sure yet about what you need and money is a concern.

Choosing the Right Amount of RAM

Similar to choosing the right CPU configurations, choosing the amount of memory for your machine involves several factors and depends on the type of jobs you are running.

If your jobs require high memory usage most of the time, your server constantly hits its memory limits and starts swapping to disk. Then it would help if you had a server with a larger memory configuration.

But before scaling your servers, make sure your jobs clean after themselves. Given that a Laravel worker uses a single daemon process to run all jobs picked up by that worker, you have to ensure you free the memory used by each job.

For example; if you are caching values in a static array or a property of a singleton, you should clean that cache when you're done:

```
// Storing items in a static array.
SomeClass::$property[] = $item;

// Flushing the cache when we're done.
SomeClass::$property = [];
```

Proper coding practices will ensure the worker doesn't consume all the memory allocated. Because if this happens, the worker will exit with a Fatal error. This immediate shutdown could leave your system corrupted if it occurs in the middle of processing a job.

To avoid this, Laravel limits the memory consumption to 128MB for each worker process. You can change the limit by using the `--memory` option:

```
php artisan queue:work --memory=256
```

> **Warning:** That worker-level limit should be lower than the `memory_limit` set in your php.ini file of PHP CLI.

After running each job, the worker will check the amount of memory used by the PHP process and terminate the script if it exceeds the limit.

Avoiding Memory Leaks

We didn't have to worry about memory leaks in most of our PHP work. Simple because PHP runs in a shared-nothing manner. The application will get a fresh state for every request, and all memory consumed by the previous process will be released back.

However, inside the worker, all jobs share the same memory space. So avoiding memory leaks can be a bit challenging. Over time, some references will pile up in memory and will cause the process to crash. The solution is easy, though, shut down the workers more often.

Laravel's `queue:work` command has a `--max-time` option that you can use to define the lifetime of a specific worker. There's also the `--max-jobs' option that limits the number of jobs the worker may process:

```
php artisan queue:work --max-jobs=1000 --max-time=3600
```

This worker will automatically shut down after processing 1000 jobs or running for an hour.

> **Notice:** After processing each job, the worker will check if it exceeded `max-jobs` or `max-time` and then decide between exiting or picking up more jobs. The worker will not exit while in the middle of processing a job.

Also, if you have a specific job that you know will consume a lot of memory and you want to free up this memory after processing it, you may signal the worker to exit by setting the `$shouldQuit` worker property to `true` at the end of the `handle()` method:

```
public function handle()
{
    // Run the job logic.

    app('queue.worker')->shouldQuit  = true;
```

```
}
```

Now this worker will shut down during the cleaning phase.

Using these options, along with the `--max-memory` option, you can make your workers restart and get a fresh memory state. How are these shutdown processes start again automatically? That's what we're going to discuss in the next chapter.

Keeping the Workers Running

Throughout the book, we've discussed starting workers by running the `queue:work` in the command line interface. We've also talked about restarting workers by shutting them down. This might have given you the notion that you need to have a 24/7 on-call engineer monitoring the workers and starting them when they shut down. Luckily, that's not the case.

Let me introduce you to *Process Control Systems*. This concept is borrowed from industry production lines during manufacturing. A process control system monitors the process by reading signals from different sensors and meters and taking actions based on the collected data.

In Unix-like operating systems, one of the most popular process control systems is Supervisor. It includes a server component (`supervisord`) that can start and monitor several processes and a command line tool (`supervisorctl`) that we can use to control the server.

To install Supervisor on your machine, you may run these commands:

```
sudo apt-get install supervisor
sudo service supervisor restart
```

Starting Workers

Now let's look into configuring Supervisor to start and monitor 4 worker processes. First, we need to create a file inside the `/etc/supervisor/conf.d` directory with this content:

```
[program:default-workers]
command=php artisan queue:work

process_name=%(program_name)s_%(process_num)02d
```

```
autostart=true
autorestart=true
stopasgroup=true
user=forge
numprocs=4
stdout_logfile=/home/forge/.forge/default-workers.log
stopwaitsecs=3600
```

We will store this file under the name `default-workers.conf`.

Now let's signal Supervisor to re-read the configuration files so it can see the new file we just added:

```
supervisorctl reread
supervisorctl update
```

Finally, let's tell Supervisor to start our program and monitor it:

```
supervisorctl start default-workers:*
```

Once this command runs, Supervisor will start four processes that run the `php artisan queue:work` command, monitor them and restart any of them when they exit.

Supervisor Configurations

Now let's look at a breakdown of the configuration keys from the above example:

`[program:default-workers]`:

This is the name of our program. We will use this name to identify the program in our communication with Supervisor:

`command` and `numprocs`:

When Supervisor starts a program with the above configurations, it's going to create a process group with four processes that run the given command:

```
php artisan queue:work
```

`autostart`, `autorestart`, and `user`:

Each process in this group will be started when Supervisor starts and auto-restarted if it exits. The `forge` OS user will run the processes.

`stopasgroup`:

If we explicitly instruct Supervisor to stop this group, the signal will be sent to every process in the group.

`stopwaitsecs`:

This is the time Supervisor will give to a process between sending the stop signal and forcefully terminating it.

Stopping Workers

There are cases where we need the workers to stop running. For example, if we simply kill the worker processes using the `KILL` command, Supervisor will start them back.

To tell Supervisor to shut down the workers and not start them back, we may use the `supervisorctl stop` command:

```
supervisorctl stop default-workers:*
```

We can also stop all workers on our machine using:

```
supervisorctl stop all
```

Running `supervisorctl stop` will instruct Supervisor to send a `SIGTERM` signal to all the workers under the specified program. And as we discussed earlier, when workers receive `SIGTERM`, they will shut down gracefully. Meaning they will finish any job being processed first before shutting down.

Given the configurations from above, Supervisor will give each worker 3600 seconds (`stopwaitsecs`) to exit. If a worker doesn't exit during this time, Supervisor will force it to exit by sending a `SIGKILL`, which is something we should avoid.

Therefore, we should set the value of `stopwaitsecs` to be longer than our longest-running job. That way, if the worker receives a `SIGTERM` signal from Supervisor, it'll be able to finish

running that job and exit gracefully before Supervisor terminates it.

Now, if we want to start those stopped workers back, we have to run `supervisorctl start` again:

```
supervisorctl start default-workers:*
```

Scalability of The Queue System

Scalability is all about handling an increase in demand. A scalable web application can serve more clients without sacrificing performance or breaking the bank. Laravel queues help with that by allowing the web application to send tasks to be done in the background. That way, it can respond faster to client requests and thus serve more of them.

But when we talk about the scalability of the queue system, we aren't usually concerned about serving more clients. Instead, we're mainly concerned about the *Average Wait Time* (AWT).

Similar to how the operating system tries its best to lower the average wait time of processes waiting for CPU time, our system should try its best to reduce the average wait time of jobs in the queue.

Besides doing performance tuning to our job logic to make it run faster, we may need to do one or more of the following:

1. Add or remove machines.
2. Add or remove workers on the same machine.
3. Allocate more or fewer workers to a certain queue.

These scaling activities can be performed based on different metrics, like the queue size, expected time to clear the queue, a fixed schedule, or a change in priorities.

Adding machines to your cluster is the most expensive of these options. Therefore, we should only consider it if we are reaching a point where all our machines are running at 100%, while time-sensitive jobs are waiting—for a long time—in the queue.

In such a case, we'll need to launch a new machine, install and configure the Laravel application and start several workers via Supervisor. We may write a script to automate this process if needed or use a service like AWS Auto Scaling groups to do this for us.

Running More Workers

On the other hand, adding workers won't increase the cost of running our system. We are using the same machine(s) we have to speed up the processing of specific queues.

When adding new workers to an existing machine, we should remember that workers compete with other processes on the machine for CPU time and memory. Therefore, by adding new workers, other processes may experience performance degradation. We don't want that for processes that run our databases, cache instances, web request handlers, etc...

However, in some instances, it is acceptable to increase the workers' share of CPU time and decrease the web handlers' share to run nightly jobs. For example, some application developers delay processing non-time-sensitive jobs during periods where the web traffic is low.

If we know precisely when we need to add more workers and when they are no longer required, we can use CRON to start a supervisor process group on a fixed schedule and stop it when the workers are no longer needed.

For example, let's add a new Supervisor configuration at `/etc/supervisor/conf.d/extra-workers.conf`:

```
[program:extra-workers]
command=php artisan queue:work notifications --queue=orders,notifications

process_name=%(program_name)s_%(process_num)02d
autostart=false
autorestart=true
stopasgroup=true
user=forge
numprocs=3
stdout_logfile=/home/forge/.forge/extra-workers.log
stopwaitsecs=3600
```

This process group will start three worker processes that consume jobs from the `orders` and `notifications` queues. We've set `autostart=false` so it doesn't start automatically when we restart the server, CRON will take care of starting and stopping this group.

To add the new group to Supervisor's memory, we need to run the following commands:

```
supervisorctl reread
supervisorctl update
```

Now let's configure our CRON jobs to start this group at 7 am and stop it at 11 am:

```
0 7 * * * root supervisorctl start extra-workers:*

0 11 * * * root supervisorctl stop extra-workers:*
```

> **Notice:** CRON is configured by updating the `/etc/crontab` file.

Between 7 am and 11 am, Supervisor will start these three worker processes for us and keep them running.

During the time these three extra workers are running, you may notice that your average concurrent database connections are increasing. That means your database will spawn more threads that will utilize more CPU time.

Each of these extra workers will require its database connection to be able to run queries. So you must keep this demand in mind. You don't want to start more workers to consume the queue faster, only to find that the database is choking and slowing everything down.

Balancing Based on Workload

The most common event that may require a scaling action is increased workload. Most of the time, this means the size of a particular queue is growing, and we need to prioritize it for most active workers.

We've seen in a previous challenge that we can control which queues workers are consuming jobs from dynamically, using the `Worker::popUsing` method. We can use this knowledge to check the size of a certain important queue and allocate more workers to it when it's busy.

But first, we need to give our workers unique names to identify them within the `popUsing()` method. So using Supervisor, let's create three programs that run three commands:

```
command=php artisan queue:work --name=orders --queue=orders,notifications
numprocs=4
```

```
command=php artisan queue:work --name=notifications --queue=notifications,orders
numprocs=1
```

```
command=php artisan queue:work --name=spare --queue=notifications,orders
numprocs=3
```

These three programs start 10 workers:

1. 5 workers with the name `orders` and priorities `orders,notifications`.
2. 1 worker with the name `notifications` and priorities `notifications,orders`.
3. 4 workers with the name `spare` and priorities `notifications,orders`.

Figure 3-12 shows our workers in their initial setup. Four workers (blue) prioritize the `orders` queue, and four (green) prioritize the `notifications` queue.

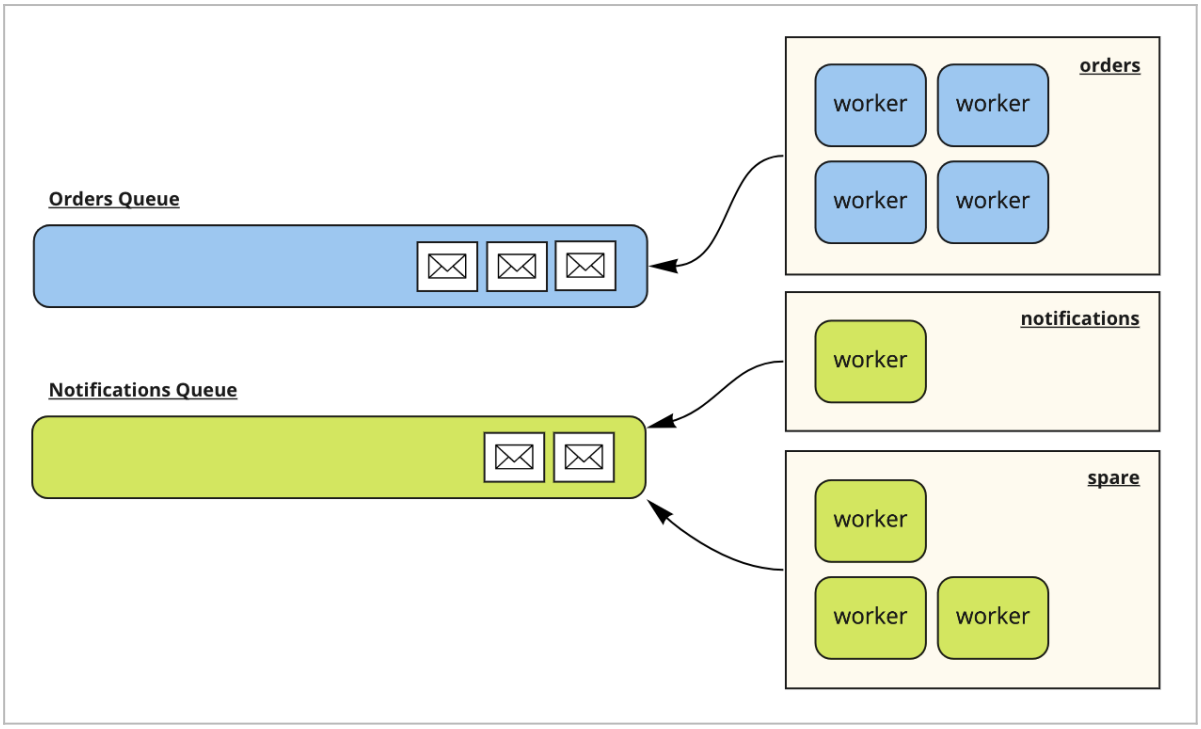

Figure 3-12. Our worker groups.

If the `notifications` queue becomes empty, all eight workers will process jobs from the `orders` queue. This ensures our workers don't stay idle while any of the queues are busy.

Now let's look into how we may use the `Worker::popUsing` method to force all workers with the name `spare` to consume jobs from the orders queue if it exceeds a certain threshold:

```
Worker::popUsing('spare', function ($pop, $queues) {
    $consumableQueues = explode(',', $queues);

    if (Queue::size('orders') > 30) {
        $consumableQueues = ['orders'];
    }

    foreach ($consumableQueues as $queue) {
        if ($job = $pop($queue)) {
            return $job;
        }
    }
});
```

Here we use the `Queue::size()` method to find the number of jobs currently in the queue. If that number is over 30, we update the `$consumableQueues` array to include only the `orders` queue. That way, the three spare workers will consume jobs from the `orders` queue only.

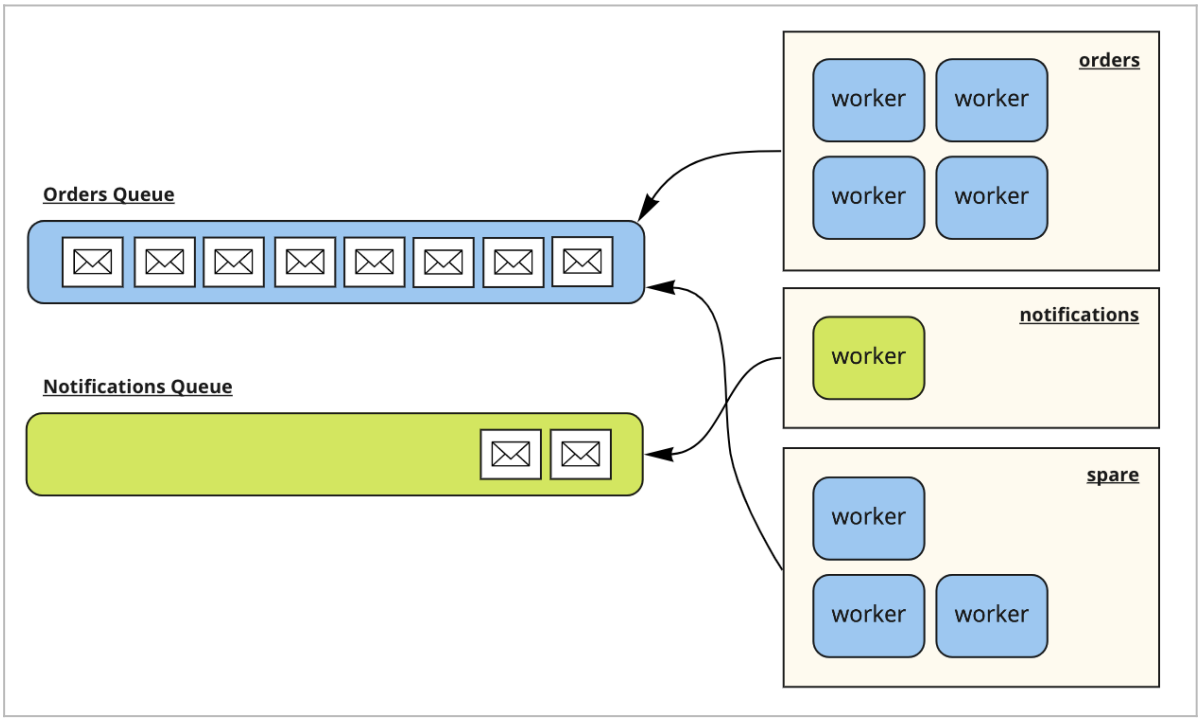

Figure 3-13. The three spare workers move to consume the orders queue.

When the size of the `orders` queue goes back to 30 or below, the `$consumableQueues` array will include the original setup, which has the `notifications` queue as priority and then the `orders` queue.

But now we're calling the `Queue::size()` method on each iteration of the worker infinite

loop. This is an overhead we want to avoid. And to do that, we'll use the `array` cache store to limit checking the queue size to once every 10 seconds:

```
Worker::popUsing('spare', function ($pop, $queues) {
    $consumableQueues = explode(',', $queues);
    $cache = Cache::store('array');

    if (is_null($size = $cache->get('size'))) {
        $cache->put('size',
            $size = Queue::size('orders'),
            10
        );
    }

    if ($size > 30) {
        $consumableQueues = ['orders'];
    }

    foreach ($consumableQueues as $queue) {
        if ($job = $pop($queue)) {
            return $job;
        }
    }
});
```

Here we check if the size is not stored in the cache already, read it from the `Queue:size()` method, and put it in the cache with a 10-second expiration.

Now on the next iteration, the `Cache::get()` method will return the last read queue size and won't call `Queue::size()` again until the key expires.

Scaling With Laravel Horizon

In the previous section, we saw how we could use the `Worker::popUsing()` function to balance workers' allocation based on workload. However, suppose we're using the Redis queue driver. In that case, Laravel provides a package called Horizon that has a friendly dashboard for displaying live metrics and a built-in auto balancer that supports different balancing strategies.

So, let's look into it.

Starting and Monitoring the Horizon Process

Instead of starting multiple workers, we only need to start the Horizon process:

```
php artisan horizon
```

We'll still need to use Supervisor to keep the Horizon process running at all times:

```
[program:horizon]
command=php artisan horizon

process_name=%(program_name)s
autostart=true
autorestart=true
stopasgroup=true
user=forge
stdout_logfile=/home/forge/.forge/notifications-workers.log
stopwaitsecs=3600
```

These configurations should be stored in a `/etc/supervisor/conf.d/horizon.conf` file.

> **Notice:** We only need a single Horizon process running. Therefore, no `numprocs` attribute is required for the Supervisor configuration file.

Basic Horizon Configurations

When we install Horizon in our application, a `horizon.php` configuration file will be published in the `config` directory. At the bottom of that file, we'll see an `environments` array.

Let's configure our `production` environment:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['deployments', 'notifications'],
            'balance' => 'off',
```

```
            'processes' => 10,
            'tries' => 1,
        ],
    ],
]
```

When Horizon starts with the given configurations, it will start 10 `queue:work` processes that consume jobs from the `deployments` and `notifications` queues. These worker processes will look like this:

```
artisan horizon:work redis --name=default
                    --supervisor=host-tdjk:supervisor-1
                    --backoff=0
                    --memory=128
                    --queue=deployments,notifications
                    --sleep=3
                    --timeout=60
                    --tries=1
```

While Supervisor ensures the Horizon process is running at all times, Horizon will ensure all ten worker processes are running at all times. If one process exits, Horizon will start it back again.

We can control how the workers are configured by updating the configuration keys for the horizon supervisor. For example, we can configure the `backoff` for all workers to 3 seconds by configuring a `backoff` key:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            // ...
            'backoff' => 3,
            // ...
        ],
    ],
]
```

We can also have multiple horizon supervisors running, each with different configurations:

```
'environments' => [
    'production' => [
```

```
        'deployments' => [
            // ...
            'timeout' => 300,
            'processes' => 7,
            'queue' => 'deployments'
            // ...
        ],

        'notifications' => [
            // ...
            'timeout' => 60,
            'processes' => 3,
            'queue' => 'notifications'
            // ...
        ],
    ],
]
```

Here, Horizon will start seven worker processes that consume jobs from the `deployments` queue, and each has a timeout of 300 seconds. In addition to 3 other worker processes that consume jobs from the `notifications` queue with a timeout of 60 seconds.

The Simple Balancing Strategy

Let's configure the balancing strategy to `simple` instead of `off`:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'connection' => 'redis',
            'queue' => ['deployments', 'notifications'],
            'balance' => 'simple',
            'processes' => 10,
            'tries' => 1,
        ],
    ],
]
```

Now Horizon will start five worker processes that consume jobs from the `deployments` queue and another five processes that consume jobs from the `notifications` queue. Using the `simple` strategy, Horizon will divide the number of processes on the queues the supervisor is configured to run.

Horizon will also allocate more workers to the queue with higher priority:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'queue' => ['deployments', 'notifications'],
            'balance' => 'simple',
            'processes' => 5,
        ],
    ],
]
```

With `processes` set to `5`, Horizon will start three workers for the `deployments` queue and two for the `notifications` queue.

The Auto Balancing Strategy

The most powerful balancing strategy Horizon provides is the `auto` strategy:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'queue' => ['deployments', 'notifications'],
            'balance' => 'auto',
            'min_processes' => 1,
            'max_processes' => 10
        ],
    ],
]
```

Using this strategy will configure Horizon to start and stop processes based on how busy each queue is. For that to work, we need to set the `min_processes` and `max_processes` configuration options.

In the above example configuration, Horizon will have a minimum of 1 worker process for each queue and a maximum of 10 processes in total.

If the `deployments` queue is busy while the `notifications` queue is empty, Horizon will allocate nine worker processes to `deployments` and a single worker to `notifications`.

Horizon decides the number of workers to allocate based on the expected time to clear per

queue. So if the jobs in the `deployments` queue take more time to run while jobs in the `notifications` queue run instantly, even if the `notifications` queue has more jobs than the `deployments` queue, Horizon will still allocate more workers to the `deployments` queue since it has the higher time-to-clear.

The Rate of Balancing

By default, Horizon adds or removes a single worker every 3 seconds. That means it takes 9 seconds to balance the pool from 5 to 8 workers.

We can control the rate of balancing by adjusting the `balanceMaxShift` and `balanceCooldown` configuration options:

```
'environments' => [
    'production' => [
        'supervisor-1' => [
            'queue' => ['deployments', 'notifications'],
            'balance' => 'auto',
            'min_processes' => 1,
            'max_processes' => 10,
            'balanceMaxShift' => 3,
            'balanceCooldown' => 1
        ],
    ],
]
```

With `balanceMaxShift` equals three and `balanceCooldown` equals one, Horizon will add or remove three workers every second while it balances the pool.

> **Notice:** Controlling the balancing rate is only possible starting Horizon v5.0.

Choosing The Right Queue Driver

Laravel gives us several options for storage drivers that we can use for our queued jobs; there's the database driver, the Redis driver, and the SQS driver. And in most cases, deciding which driver to use depends on how easy we can install the driver and keep it reliably running.

We all know how to work with databases, which makes the database driver the first choice for most of us. Not just because we can run it on any machine, but also because it gives us very high visibility. Using simple queries, we can see what's happening inside the queue.

The database driver is a perfect option for projects with limited workloads. However, it's not ideal for high throughput environments with thousands of jobs dispatched every hour. Each dispatch is a write operation, which translates to a disk I\O operation. An increased load means slower write operations and longer AWT for our jobs.

Switching to a different driver is easy, so we can start with the database driver for small projects and switch to a more advanced one as the traffic increases.

The Redis Queue Driver

Redis is an in-memory key-value store that is single-threaded from the user's point of view. This means two things:

1. It's extremely fast.
2. It guarantees consistency.

Redis stores all data in the machine RAM, which is super fast when compared to secondary storage like a traditional relational database.

Also, since it's single-threaded, only a single command is executed at any given time. That means our reads & writes are always consistent, even under very high traffic. If two workers, for example, tried to dequeue messages simultaneously, there's no way both workers will end up getting the same message.

Communication with Redis occurs in the form of commands:

```
// Push a message to the invoices queue
RPUSH invoices "Send Invoice #1"

// Pop a message from the invoices queue
LPOP invoices
```

Knowing that Redis stores all our jobs in the machine RAM, there are a few things to consider:

1. Choose a memory-optimized machine to host Redis.
2. Ensure the payload of our jobs is as small as possible
3. Ensure we have enough workers to process jobs as fast as possible.

A memory-optimized machine will give us better performance in reading and writing to the machine RAM. But, on the other hand, limiting the size of our jobs and speeding up the processing rate means we'll efficiently use the allocated memory.

By default, Redis comes with zero memory limits on 64-bit systems. That means it can gradually eat up all the machine's available memory and start swapping to disk, which leads to a significant increase in latency. That's why keeping the memory in check is essential.

Moreover, we can configure Redis to have limits on the maximum memory it can use by adjusting the `maxmemory` configuration parameter. When Redis reaches this limit, it'll start returning an error on write operations. That means job pushing operations will fail, enough noise to warn us about the situation.

On the other hand, given that Redis is single-threaded, we need to watch out for slow commands. A single slow command may block all clients from communicating with Redis for a significant time. This includes clients initiated from web servers (pushing new jobs) and clients initiated from workers (poping jobs).

Laravel's commands for pushing and popping jobs are very quick. However, dispatching a large batch of jobs may lead to a slow command that blocks Redis until all jobs are persisted. Therefore, it's generally recommended that we limit the number of jobs dispatched in a single command to avoid that.

We can also monitor for slow commands by running the `SLOWLOG` Redis command:

```
redis-cli SLOWLOG GET 10
```

The command above will return the ten most recent slow commands. Also, showing the timestamp when they were executed, the duration, the arguments, and the client IP address.

By default, Redis will log a command that takes more than 10000 microseconds to complete as a slow command. However, we can change this value by adjusting the `slowlog-log-slower-than` configuration parameter.

Persisting Redis Data to Disk

Memory is volatile. If the server restarts while there are jobs in our queues, those jobs will be lost. That's why, by default, Redis stores a snapshot of its data on disk when certain conditions are met. It uses the snapshot to re-create the in-memory store after a restart event.

These snapshots are taken under these conditions:

1. After one hour, if at least one change was performed.
2. After 5 minutes, if at least 100 changes were performed.
3. After 60 seconds, if at least 10000 changes were performed.

> **Notice:** We can configure these conditions by adjusting the `save` configuration parameter.

Snapshotting can be of great use for disaster recovery, but it doesn't guarantee all the data will be restored. For example, all data written between the time the latest snapshot was taken and the time of the restart will be lost. Depending on the type of work we perform in the queue, this may or may not be a big deal. If it is, we should consider the *Append-only* persistence strategy.

This strategy works by appending every write command to an append-only file. Then, when Redis restarts, it'll use this file to rebuild the data in memory. To enable this, we may adjust the `appendonly` configuration parameter and set it to `yes`.

Using a Separate Database for Queues

In Laravel, we can use Redis as a cache store. While this is generally ok, it puts us at risk. If we decide to flush the cache for any reason, our jobs will be flushed with it.

When Laravel flushes the Redis cache, it uses the `FLUSHDB` command, which deletes everything in the Redis database.

For that reason, it's generally recommended that we use a separate Redis instance for our queues. Or, if we have to use a single instance, we must configure our queues to use a dedicated Redis connection that interacts with a dedicated database.

To configure a new Redis connection, we may adjust the `database.php` configuration file:

```
[
    'redis' => [
        //...

        'queue' => [
            // ...
            'database' => env('REDIS_QUEUE_DB', '3'),
```

```
        ],

    ],
];
```

And then, we can adjust the `redis` queue connection to use this Redis connection in the `queue.php` configuration file:

```
[
    'connections' => [
        // ...

        'redis' => [
            // ...
            'connection' => 'queue',
        ],
    ],
];
```

Now jobs will be stored in database #3 in our Redis instance. So Flushing the cache will not affect the jobs anymore.

Poping Redis jobs using long polling

When poping jobs from Redis, the server will return `nil` if the queue is empty. And as we've explained in a previous section, the worker will sleep for a few seconds, 3 seconds by default, before trying to pop jobs again.

If our application has time-sensitive jobs and a three-second wait might not be acceptable, we may reduce the sleep time to `0` by adjusting the worker's `--sleep` option. That way, the worker will keep polling constantly until a job becomes available (Figure 3-14).

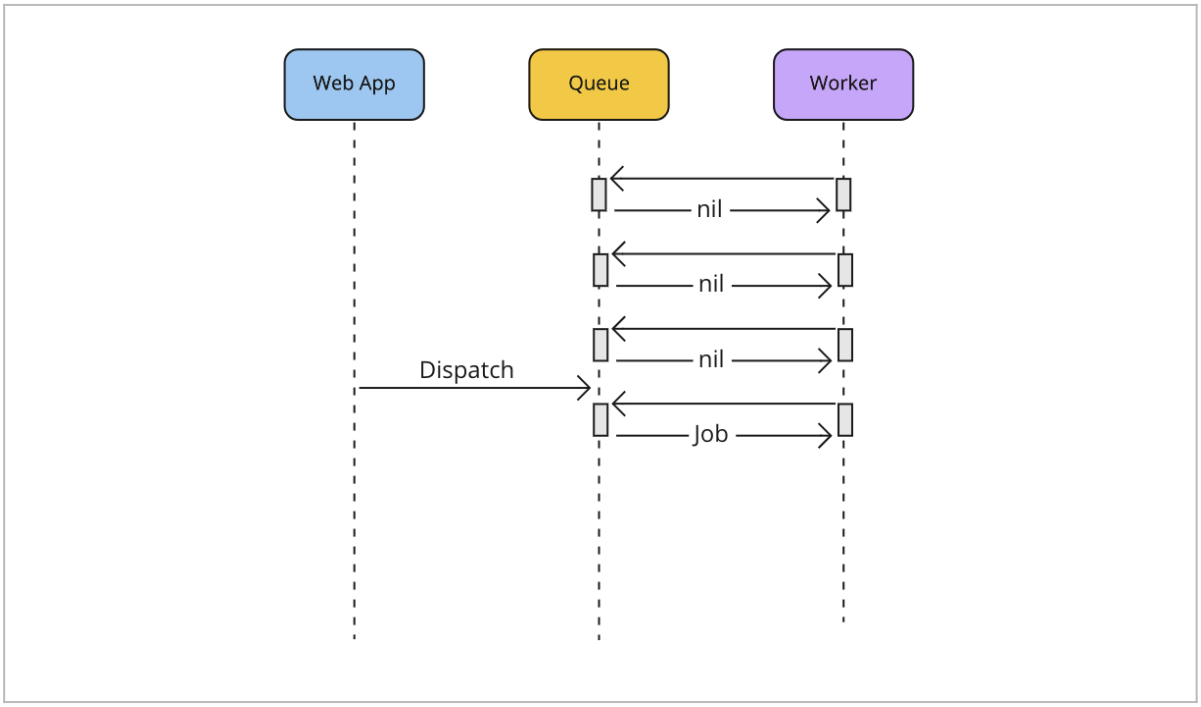

Figure 3-14. Short polling for jobs.

Each pop request is a Redis command, and having multiple workers hammering the Redis instance with pop requests means push requests may get delayed.

In such an application, a better option would be long polling (Figure 3-15). With this method, the Redis server will block the client connection until a job becomes available. During this time, Redis itself won't be blocked and will be able to handle other commands.

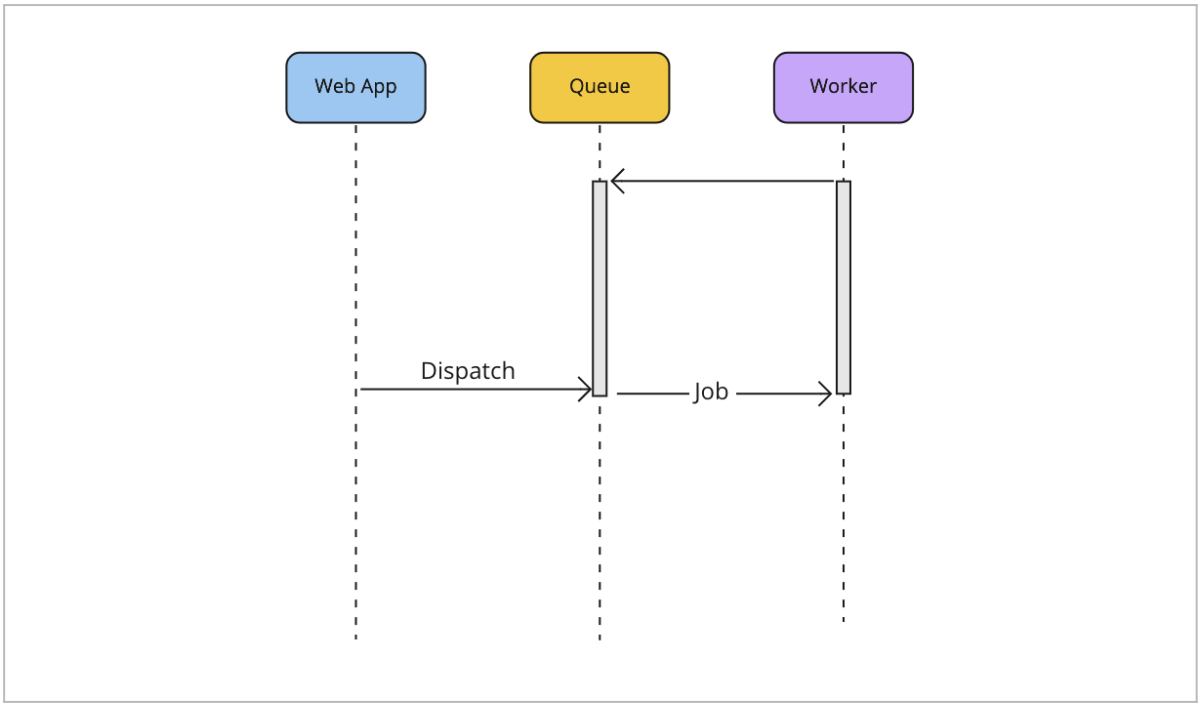

Figure 3-15. long polling for jobs.

Long polling guarantees jobs will be popped as soon as they become available, without sending too many Redis commands or utilizing more CPU time on the worker machine.

During the time the worker is blocked, waiting for a job, the PHP process will enter a `WAIT` state, and the operating system scheduler will allocate CPU time to other processes. Maybe other workers that are consuming jobs from a busy queue. Which means better utilization of our system resources.

To switch to long polling, you'll need to update the `block_for` attribute in the `redis` connection inside the `queue.php` configuration file:

```
[
    'connections' => [
        // ...

        'redis' => [
            // ...
            'block_for' => 10,
        ],
    ],
];
```

With the configuration above, Redis will keep the client connection open for 10 seconds until a job is poped.

The SQS Queue Driver

SQS is a managed queue store in the AWS cloud. If our application is hosted in AWS, communication between the application and SQS will be super fast. We also won't have to worry about persistence or recovery. AWS promises to handle all this for us.

However, there are several issues with SQS that we need to have in mind while making the decision:

1. The maximum job payload size is 256 KB.
2. The maximum delay you can use is 15 minutes.
3. The maximum life span of a job is 12 hours.
4. SQS doesn't guarantee our jobs will be dequeued only once.

Dealing with the payload size limit and the delay/backoff limit is manageable. However, dealing with the at-least-once delivery limitation can be a bit tricky.

In most jobs, it's not disastrous to have the job running multiple times. No harm in sending a user two welcome emails, for example. However, if specific jobs must only run once, we need to design those jobs to be idempotent.

> **Notice:** Designing idempotent jobs is a good practice while using any queue driver, not just SQS.

Another thing with SQS is that we cannot set a `retry_after` value in our `queue.php` configuration file. Instead, we'll need to go to the AWS console and configure the "Default Visibility Timeout" of the queue.

This value represents the number of seconds SQS will mark the job as reserved when a worker picks it up. After the worker runs the job, it should be deleted or released back to the queue to be retried.

The maximum time we can keep a job in the queue is 12 hours from when SQS receives the job. Therefore, if we need to keep releasing the job back to the queue for more than 12 hours, we will have to delete the message and dispatch a fresh one.

Switching Between Drivers

To switch between queue drivers, we need to follow the following steps:

1. Configure a new queue connection in the queue.php configuration file for the new driver.
2. Update our code to push new jobs to that connection.
3. Start new workers to process jobs from the new connection.
4. Wait until old workers finish processing all jobs still stored in the old connection.
5. Stop those old workers.

For example, if we are switching from the `database` queue connection to the `redis` connection. We need to update our code to push new jobs on the `redis` connection:

```
NewJob::dispatch()->onConnection('redis');
```

Then start a few workers to process jobs from the `redis` connection so end-users don't experience too much delay:

```
php artisan queue:work redis --queue=invoices --timeout=30
```

Finally, we can use `Queue:size()` to find the number of pending jobs in each queue:

```
Queue::connection('database')->size('invoices')
```

When all queues are empty, we can stop the old workers to make room for adding more workers that process jobs from the new connection.

Handling Queues on Deployments

When we change our application code or configurations and deploy, the workers won't read those changes automatically. Instead, they need to be restarted, so a fresh instance of the Laravel application is loaded with the new changes or configurations.

So when writing the deployment script, we need may run the `queue:restart` command after pulling the new changes from git and restarting PHP-fpm:

```
git pull origin master

composer install

sudo -S service php8.1-fpm reload

php artisan queue:restart
```

After `php-fpm` restarts, the web visitors of our application will start using the new code while workers are still running on older code. Eventually, those workers will restart gracefully and run the new code.

Restarting Horizon

Similar to restarting regular worker processes, we can signal Horizon's master supervisor to terminate all worker processes by using the following command:

```
php artisan horizon:terminate
```

However, to ensure our jobs won't be interrupted, we need to make sure of the following:

1. The Horizon supervisors' `timeout` value is greater than the number of seconds consumed by the longest-running job.
2. The job-specific `timeout` is shorter than the timeout value of the Horizon supervisor.
3. If we're using Supervisor as our process control system, we need to ensure the value of `stopwaitsecs` is greater than the number of seconds consumed by the longest-running job.

With this correctly configured, Supervisor will wait for the Horizon process to terminate and won't force-terminate it after `stopwaitsecs` passes. Likewise, Horizon will wait for the longest job to finish running and won't force-terminate after the timeout value expires.

Dealing With Migrations

When we send a restart signal to the workers, using the `queue:restart` command, some of them may not restart right away; they'll wait for a job in hand to be processed before shutting down.

If our deployment includes migrations that'll change the database schema, workers still using the old code may fail in the middle of running their last job. Because the database schema suddenly changed.

To avoid this, we need to signal the workers to shut down before we start the deployment and bring them back up after the deployment finishes.

To do that, we may use the `supervisorctl stop` command. This command will block the execution of the script until all workers are shutdown:

```
sudo supervisorctl stop default-workers:*

git pull origin master

composer install

sudo -S service php8.1-fpm reload

php artisan migrate --force

sudo supervisorctl start default-workers:*
```

> **Warning:** Make sure the deployment's system user can run the `supervisorctl` command as `sudo`.

Now, the deployment won't run until all running workers are shut down.

Designing Reliable Queued Jobs

Queued jobs are PHP objects that get serialized, persisted, transferred, unserialized, and finally executed. They may run multiple times, on different machines, and they may run in parallel.

In this chapter, we're going to look into designing reliable jobs.

Making Jobs Self-contained

There's no way we can know for sure when a queued job is going to run. It may run instantly after being sent to the queue or after a few hours.

Since the state of the system may change between the time the job was dispatched and the time a worker picked it up, we need to make sure our jobs are self-contained; meaning they have everything they need to run without relying on any external system state:

```
DeployProject::dispatch(
    $site, $site->lastCommitHash()
);

class DeployProject implements ShouldQueue
{
    public function __construct(Site $site, string $commitHash)
    {
        $this->site = $site;
        $this->commitHash = $commitHash;
    }
}
```

In this example job, we could have extracted the last commit hash inside the `handle` method of the job:

```
public function handle()
{
    $commitHash = $this->site->latestCommitHash;
}
```

However, new commits may have been sent to the site repository by the time the job runs.

Suppose the purpose of this job was to deploy the latest commit, then extracting the last commit when the job runs would have made sense. However, this job deploys the last commit sent when the user manually triggers the deployment.

If the user changed the color of the "Log In" button to green, deployed, and then changed it to blue. They'd expect the deployment to give them a green button.

While designing our jobs, we must take a look at every piece of information and decide if we want this data to be independent of time or not.

Making Jobs Simple

Job objects will be serialized, JSON encoded, sent to the queue store, transferred over the network, and then converted back to an object to be executed.

A job object with very complex dependencies will consume a lot of resources while being serialized/unserialized. It will also occupy more space to be stored and will take more time to move to/from the queue store.

The job class properties should be simple data types; strings, integers, booleans, arrays, and the like. Or, if we must pass an object, it has to be a simple PHP object that has simple properties:

```
use Illuminate\Contracts\Cache\Factory;

class GenerateReport implements ShouldQueue
{
    public function __construct(Factory $cache, Report $report)
    {
        $this->cache = $cache;
        $this->report = $report;
    }
}
```

This `GenerateReport` job has two dependencies; Laravel's cache manager and the `Report` eloquent model. These two objects are complex ones. Serializing/unserializing these objects will be CPU-intensive, and storing the payload and transferring it to/from the store will be I/O-intensive.

Instead, we can rely on Laravel's service container to resolve an instance of the cache manager instead of passing it to the job:

```
class GenerateReport implements ShouldQueue
{
    public function __construct(Report $report)
    {
        $this->report = $report;
    }

    public function handle()
    {
        app(
            \Illuminate\Contracts\Cache\Factory::class
        )->get(...);
    }
}
```

Since a worker runs all jobs using a single application instance, resolving the cache singleton will be easy without much overhead.

Another thing we should do here is using the `SerializesModels` trait:

```
use Illuminate\Queue\SerializesModels;

class GenerateReport implements ShouldQueue
{
    use SerializesModels;

    public function __construct(Report $report)
    {
        $this->report = $report;
    }
}
```

While serializing this job, Eloquent models will be converted to a simple PHP object. Workers will retrieve the original model later when the job runs.

Making Jobs Light

When storing a job in Redis, it will occupy part of the Redis instance memory. Or, if we're using SQS, there's a payload size limit of 256 KB per job. In addition to these considerations, there's also the latency our job may add if it's too big to move fast on the network.

Try to make the job payload as light as possible. For example, if we have to pass a big chunk of data to the job, we should consider storing it somewhere and give a reference to the job instead.

Here are the things that affect the payload size of a job:

1. Job class properties.
2. Queued closure body.
3. Queued closure "use" variables.
4. Job Chains.
5. Encrypted jobs.

Let's take a look at this closure:

```
$invoice = Invoice::find();

dispatch(function () use ($invoice) {
    // closure body here...
});
```

To serialize this closure for storage, Laravel uses the laravel/serializable-closure library. This library reads the body of the closure and stores it as a string:

```
"function\";s:65:\"function () use ($invoice) {\n
// closure body here...\n        }\"
```

It'll also sign the serialization string using our application's key to validate the signature while unserializing the job to ensure it wasn't altered while in-store.

Variables passed to the closure are also included in the payload:

```
"use\";a:1:{s:6:\"invoice\";O:45:\"Illuminate\\Contracts\\Database\\ModelIdentifier\":4:
{s:5:\"class\";s:8:\"App\\Invoice\";s:2:\"id\";N;s:9:\"relations\";a:0:{}s:10:\"connecti
on\";N;}}s:8:\"
```

As you can see, Laravel takes care of converting an Eloquent model to an identifier automatically for us. However, if we need to pass any complex PHP objects or a giant blob of data, we should know that it'll be stored in the job payload.

If we notice the queued closure body is becoming bigger than a few lines of code, we should consider converting the job to an object form instead.

As for chains, the payload of all jobs in a chain will be stored inside the payload of the first job:

```
Bus::chain(
    new EnsureANetworkExists(),
    new EnsureNetworkHasInternetAccess(),
    new CreateDatabase()
)->dispatch();
```

For example, the payload of `EnsureNetworkHasInternetAccess` and `CreateDatabase` will be stored inside the payload of `EnsureANetworkExists` as a property named `chained`.

If a chain is too long that the payload size will get out of control, we should consider starting the chain with a few jobs and add more jobs to the chain from inside the last job:

```
public function handle
{
    $this->chained = [
        $this->serializeJob(
            new AddDatabaseUser($message)
        ),
        $this->serializeJob(
            new MigrateDatabase($message)
        ),
        ...
    ];
}
```

Making Jobs Idempotent

An idempotent job is a job that may run several times without having any negative side effects.

```
public function handle()
{
    if ($this->invoice->refunded) {
        return $this->delete();
    }

    $this->invoice->refund();
}
```

In this example job, we first check if the invoice was refunded already before attempting to refund it. That way, if the job runs multiple times, only one refund will be sent.

Making Jobs Parallelizable

Keep in mind that multiple jobs can be running simultaneously, which may lead to race conditions when they try to read/write from/to a single resource.

We may use cache locks to prevent multiple jobs from running concurrently if concurrency will negatively affect the system's overall state.

We may also use funnelling to limit the number of concurrent executions of specific jobs.

Managing The State

When we start a worker process, a single application instance is booted up and stored in memory. All jobs processed by this worker will be using that same instance and the same shared memory. That means values stored in static properties will be re-used between jobs. The same goes for container bindings.

While writing our queued jobs, we need to keep in mind the state of the application when each job runs. For example, if our default system locale is `en` while we need to set the locale to something different when running a specific job, we may pass that locale to this job as a constructor parameter:

```
class SendReport
{
    public function __construct($report, $locale)
    {
        // ...
    }
```

```
    public function handle()
    {
        app()->setLocale($this->locale);
    }
}
```

Now, all we need is to dispatch the job with the locale defined:

```
SendReport::dispatch($report, 'de');
```

While the job is being processed, it will set the application locale to `de` instead of the default locale. Now here's the problem; when the worker picks up a job that assumes the default locale is `en`, it will process it with the state change made by our `SendReport` job. In other words, the default locale will be `de` after our `SendReport` job runs and will be applied to all jobs in the future.

To deal with such a problem, we need to ensure the locale is set back to the default after the `SendReport` job finishes processing:

```
class SendReport
{
    public function __construct($report, $locale)
    {
        // ...
    }

    public function handle()
    {
        app()->setLocale($this->locale);

        // Run the job logic...

        app()->setLocale('en');
    }
}
```

We also need to ensure the state is reset if the job fails. For that, we're going to implement a `failed()` method:

```
class SendReport
{
```

```
    public function __construct($report, $locale)
    {
        // ...
    }

    public function handle()
    {
        app()->setLocale($this->locale);

        // Run the job logic...

        app()->setLocale('en');
    }

    public function failed($e)
    {
        app()->setLocale('en');
    }
}
```

Same goes for any static properties we set:

```
class SendReport
{
    public function handle()
    {
        Vendor::$name = $this->vendor->name;

        // Run the job logic...

        Vendor::$name = 'Laravel';
    }

    public function failed($e)
    {
        Vendor::$name = 'Laravel';
    }
}
```

Using Job Middleware

If the application jobs are constantly making changes to the application state, it might be a good idea to reset the state to the default in a job middleware to avoid code duplication:

```
class ResetStateMiddleware
{
    public function handle($job, $next)
    {
        app()->setLocale('en');

        Vendor::$name = 'Laravel';

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

Now when this middleware is used within a job, it will ensure the defaults are set to their original values before the job code runs:

```
class SendReport
{
    public function __construct($report, $locale)
    {
        // ...
    }

    public function handle()
    {
        // ...
    }

    public function middleware()
    {
        return [
            new ResetStateMiddleware()
        ];
    }
}
```

Using Job Events

When workers pick a job up, the job instance will be resolved from the serialized payload, passed down to the middleware, and executed. However, sometimes we need to change the state before the job instance is resolved. A common use case for this is setting the tenant database connection configuration so that the right connection would be set when a serialized model is being resolved.

Luckily Laravel workers fire several events during the fetching and processing jobs. With that in mind, we can listen to the `JobProcessing` event—which is fired right after the job is picked up by the worker and before it's resolved—and set the database connection configurations:

```
class TenancyProvider extends ServiceProvider
{
    public function boot()
    {
        Event::listen(JobProcessing::class, function ($event){
            // Set the connection configurations.
        });
    }
}
```

Since the event is fired before resolving the job instance, we won't have access to the job class properties. We must store the information we need inside the raw job payload while dispatching it. To do that, we're going to see a payload creation callback on the queue:

```
class TenancyProvider extends ServiceProvider
{
    public function boot()
    {
        Queue::createPayloadUsing(function () {
            return ['tenant_id' => Tenant::get()->id];
        });

        Event::listen(JobProcessing::class, function ($event){
            $tenant = Tenant::find(
                $event->job->payload()['tenant_id']
            );

            $tenant->configureDatabaseConnection();

            $tenant->setAsCurrent();
        });
    }
}
```

Using `Queue::createPayloadUsing`, we instructed Laravel to attach the `tenant_id` of the current tenant to the job payload every time a job is dispatched. Then inside the listener, we extract this `tenant_id` from the payload and use it to configure the tenant database

connection.

Dealing With Failure

A queue system involves three main players:

1. The producer
2. The consumer
3. The store

The producer enqueues messages, the consumer dequeues and processes them, and the store keeps messages until they're processed. All three players can live on the same machine or completely different machines.

Things can go wrong with any of these players or their communication channels. For example, the Redis store can experience downtime causing errors while the producer is enqueueing jobs, or the worker process crashes while in the middle of processing a job. Part of the challenge when dealing with queues is handling failure to keep our system state consistent.

Failing to Serialize a Job

The queue system can only store jobs in a string form, which means it needs to be able to convert a job instance to a string by serializing it:

```
$queuedMessage = [
    'job' => serialize(clone $job),
    'payload' => [
        // attempts, timeout, backoff, retryUntil, ...
    ]
];

return json_encode($queuedMessage);
```

Laravel serializes the job object and puts the result in an array. That array is later JSON encoded before being sent to the queue store.

If PHP fails to serialize the object or encode the array, it will throw an exception, and the job will not be sent to the queue. To eliminate this kind of failure, we need to make sure our job instances, along with all their dependencies, are serializable.

Laravel, for example, takes care of transforming eloquent models into a serializable form when we use the `Illuminate\Queue\SerializesModels` trait in our job classes. It also wraps queued closures inside an instance of `Illuminate\Queue\SerializableClosure`, which can be serialized.

It's generally good practice to keep the properties of a queued job instance as light as possible.

Failing to Send a Job

If the job was successfully serialized, the subsequent failure point could be sending that job to the queue store. Problems may happen due to a large job payload or a networking issue.

Some queue drivers have a limit for the job size; SQS, for example, has a limit of 256 KB per message. This limitation has to be put in mind while building the job class. Keep the class dependencies simple, just enough to reconstruct the state of the job when it's time for it to run.

For networking issues, on the other hand, we need to have a retry mechanism in place to ensure our jobs aren't lost if the queue store doesn't receive them. A simple way to do that is using the `retry()` helper:

```
retry(2, function() {
    GenerateReport::dispatch();
}, 5000);
```

In this example, we will attempt to dispatch the job two times and sleep 5 seconds in between before throwing the exception.

Retrying immediately is helpful when the failure is temporary. For example, if the SQS service is down and responding with 404 errors. However, if the queue store takes more than a few seconds to recover, we'll need a more advanced retry mechanism.

Dead Letter Queue

Some jobs are not critical; if dispatching the job fails, the end-user will get an error and may retry to trigger the action sometime later. However, other jobs are critical and must be sent to the queue eventually without interaction from the end-user.

A `GenerateReport` report is dispatched when the user clicks a button in the UI. If the message sending fails, the user will get an exception and can try again later. On the other

hand, a `SendOrderToSupplier` job that's sent after the order is made is triggered automatically. So if sending the job fails, we need to store it somewhere and keep retrying.

Let's implement a simple client-side dead letter queue:

> **Notice:** A dead letter is an undelivered piece of mail. A client-side dead letter queue is a queue store on the sender end where they keep queued messages that bounce.

```
$job = new SendOrderToSupplier($order);

try {
    retry(2, function () use ($job) {
        dispatch($job)->onQueue('high');
    }, 5000);
} catch (Throwable $e) {
    DB::table('dead_letter_queue')->insert([
        'message' => serialize(clone $job),
        'failed_at' => now()
    ]);
}
```

Here we store the job in a `dead_letter_queue` database table if we encounter failure while sending it. We can set up a CRON job to check the database table periodically and re-dispatch any dead letters.

```
DB::table('dead_letter_queue')->take(50)->each(function($record) {
    try {
        dispatch(
            unserialize($record->message)
        );
    } catch (Throwable $e) {
        // Keep the job in the dead letter queue to be retried later.
        return;
    }

    // Delete the job once it's successfully dispatched.
    DB::table('dead_letter_queue')->where('id', $record->id)->delete();
})
```

Failing to Retrieve Jobs

Once the jobs reach the queue store, workers start picking them up for processing. If the workers experience failure when trying to dequeue jobs, an exception will be reported, and the worker will pause for one second before retrying again.

If we're using an error tracker like Flare or Bugsnag, we'll get alerted when the workers start throwing exceptions. Based on the situation, we may need to fix the connection issue on the queue store side.

> **Notice:** If we're using the database queue driver, Laravel will exit the worker automatically if it detects a database connection error.

Failing To Run Jobs

A job may fail to run for multiple reasons:

- An exception was thrown
- Job Timed Out
- Server Crashing
- Worker Process Crashing

Each time a job fails, the number of attempts will increment. If we have a maximum number of attempts, a maximum number of exceptions, or a job expiration date configured, the job will be deleted from the queue, and workers will stop retrying it if it hits any of these limits.

We can examine these failing jobs by checking the `failed_jobs` database table. We can also retry a specific job using the `queue:retry` command:

```
php artisan queue:retry {jobId}
```

Or retry all jobs:

```
php artisan queue:retry all
```

While checking the `failed_jobs` table, we can see the exception that was thrown and caused each job to fail. However, we may see an exception that looks like this:

```
Job has been attempted too many times or run too long. The job may have previously timed
out.
```

If you see this exception, it means one of four things happened:

1. The job has timed out while running the last attempt.
2. The worker/server crashed in the middle of the last attempt.
3. The job was released back to the queue while running the last attempt.
4. The `retry_after` configuration value is less than the job timeout that a new instance of the job was started while the last attempt is still running.

To prevent the fourth scenario from happening, we need to ensure the `retry_after` connection configuration inside our `queue.php` configuration file is larger than the timeout of the longest-running job. That way, a new job instance will not start until any running instance finishes.

Dispatching Large Batches

When we dispatch a batch of jobs to the queue, Laravel creates a record for the batch in a `job_batches` database table and stores several kinds of information about the batch, including the total number of jobs and the total number of pending jobs. These numbers are also updated whenever new jobs are added to the batch (Figure 3-16).

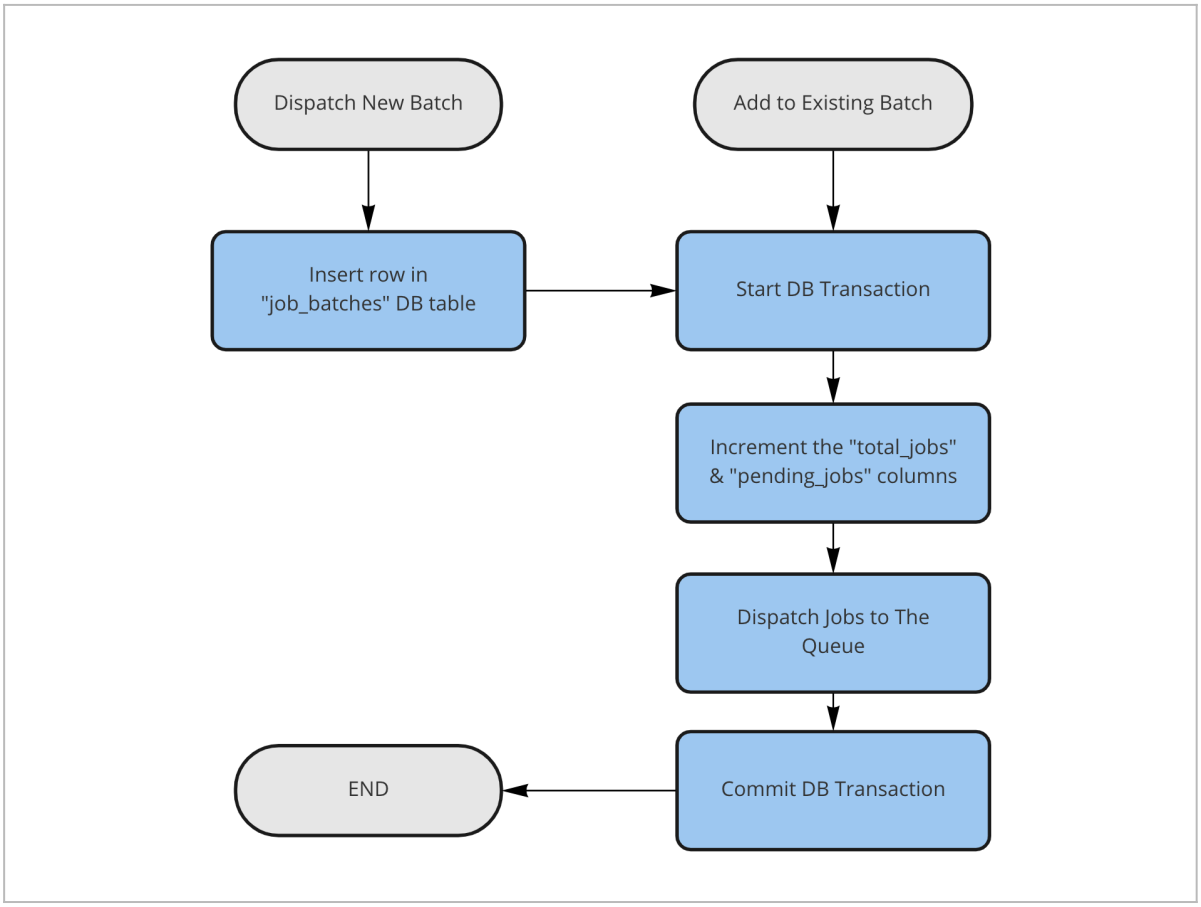

Figure 3-16. Dispatching a new batch vs. adding to an existing one.

While the database transaction is open, a lock is acquired on the batch row. During this time, other workers trying to update the row will have to wait until the lock is released. Workers try to update the row after processing a job that belongs to the batch, whether completed or failed.

When the number of jobs being dispatched is large, storing all these jobs in the queue driver may take some time. That means the database row will be locked for a more extended period, and other workers will have to wait longer (Figure 3-17).

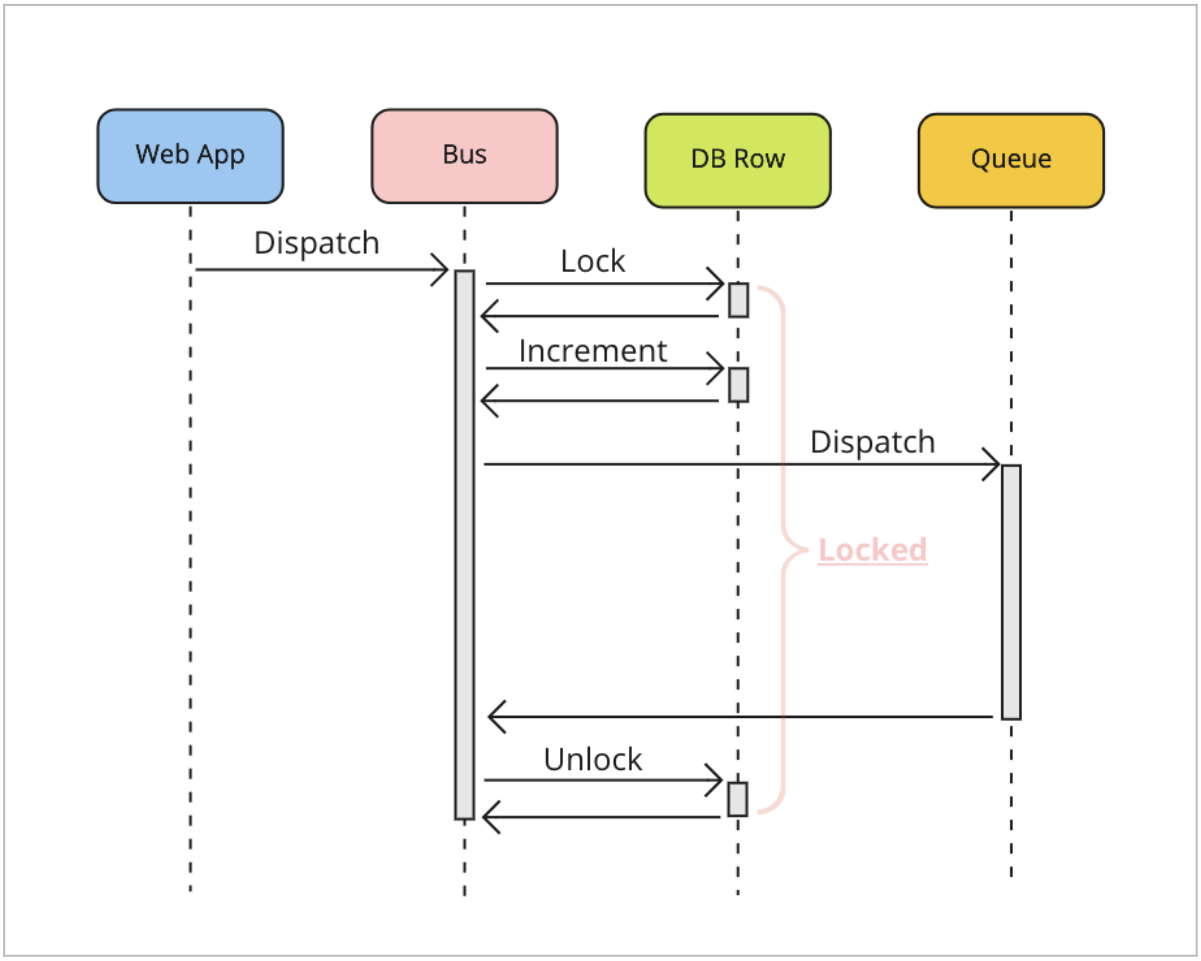

Figure 3-17. The row is locked while jobs are being dispatched.

While workers are waiting to update the batch, after processing a job, the worker timeout alarm may go off and cause the worker to shut down. When this happens, the job may be retried while it has already completed its business logic.

Also, if we're using MySQL with InnoDB as the database engine, the query will timeout if it spends 50 seconds waiting for a lock. When this happens, we'll see an error like this in our logs:

```
SQLSTATE[HY000]: General error: 1205 Lock wait timeout exceeded; try restarting the
transaction.
```

To avoid this issue, we need to dispatch jobs in small chunks to release the lock on the row inside the `job_batches` table as soon as possible. This is recommended when dispatching a new batch and when adding more jobs to an existing batch.

CHAPTER 4

Reference

In this part, you'll find a reference to all queue configurations. Including worker-level configurations, job-level configurations, connection configurations, and Horizon configurations. We'll also look into the built-in job middleware that Laravel ships with.

Worker Configurations

connection

```
php artisan queue:work redis
```

By default, workers are going to process jobs from the connection set by the `QUEUE_CONNECTION` environment variable. Using the `connection` command argument instructs the worker to use a different connection.

Connections are configured in `config/queue.php`.

queue

```
php artisan queue:work --queue=list,of,queues
```

Workers process jobs from the default queue specified in the connection's configurations in `config/queue.php`.

Using the '-- queue' command option, you can configure each worker to process jobs from a different queue. You can also configure each queue's priority based on the list's order.

name

```
php artisan queue:work --name=notifications
```

Giving your workers a name makes it easier for you to identify each worker process when investigating the running processes on your machine.

You can also use the name to customize which queues the workers must consume jobs from:

```
Worker::popUsing('notifications', function ($pop) {
    $queues = time()->atNight()
        ? ['mail', 'webhooks']
        : ['push-notifications', 'sms', 'mail', 'webhooks'];

    foreach ($queues as $queue) {
        if (! is_null($job = $pop($queue))) {
            return $job;
        }
    }
});
```

once

```
php artisan queue:work --once
```

Using the `--once` option instructs the worker to process a single job and exit. This can be useful if you need to restart the worker after each job to run jobs in a fresh state or clear the memory.

> **Warning:** Using this option increases the CPU consumption of your worker processes since an application instance will need to be bootstrapped for each job.

stop-when-empty

```
php artisan queue:work --stop-when-empty
```

This option instructs the worker to exit once it finishes all jobs in the queues assigned to it.

It comes in handy when you start workers with an autoscaling script and want those workers

to exit once the queue is clear of jobs.

max-jobs

```
php artisan queue:work --max-jobs=1000
```

This option instructs the worker to exit after processing a specified number of jobs.

max-time

```
php artisan queue:work --max-time=3600
```

This option instructs the worker to exit after running for a specified number of seconds.

force

```
php artisan queue:work --force
```

Workers will stop picking up new jobs when your application is in maintenance mode. Using the `--force` option instructs the workers to keep processing jobs even when the application is down for maintenance.

memory

```
php artisan queue:work --memory=128
```

Using the `--memory` option, you can instruct the worker to exit once the PHP process detects a specific amount of memory allocated to it.

Clearing the memory before reaching the memory limit ensures your workers keep running without issues.

sleep

```
php artisan queue:work --sleep=1
```

When no jobs are found in the queues assigned to this worker, the worker will sleep for 3 seconds by default to avoid using all available CPU cycles.

You may use the `--sleep` option to configure a different duration for the worker to sleep before picking up jobs again.

rest

```
php artisan queue:work --rest=1
```

This option specifies the number of seconds the worker should sleep between jobs. It can be used to slow down job processing.

backoff

```
php artisan queue:work --backoff=0
```

The worker uses the specified value in the `--backoff` command option as a delay when releasing failed jobs back to the queue. By default, it releases the jobs with no delay.

You can also specify an array of values to have a different delay based on how many times the job was attempted:

```
php artisan queue:work --backoff=30,60,300,900
```

timeout

```
php artisan queue:work --timeout=60
```

Using `--timeout`, you may specify the maximum number of seconds a job may run. After this duration, the worker will terminate the job and exit.

tries

```
php artisan queue:work --tries=3
```

When a job fails, it will be attempted for the number of times specified in the `--tries` option. After the job consumes all the attempts, it will be considered as failed and put in the `failed_jobs` database table for inspection.

You can configure a job to retry indefinitely by providing `0` as the number of tries.

Job Configurations

connection

```
public $connection = 'redis';
```

You can explicitly set the connection a specific job should be pushed to by using the `$connection` public property.

queue

```
public $queue = 'notifications';
```

Similarly, you can specify the queue the job should be pushed to.

backoff

When the job fails, the worker is going to use the backoff specified in the job as a delay when releasing the job back to the queue. You can configure that using a `$backoff` publish property:

```
public $backoff = 30;
```

Or a method:

```
public function backoff()
{
    return 30;
}
```

You can also set an array:

```
public $backoff = [30, 60, 300, 900];
```

timeout

```
public $timeout = 60;
```

This sets the maximum duration—in seconds—the job is allowed to run. After that time passes, the worker is going to terminate the job and exit.

tries

```
public $tries = 3;
```

When a job fails, it will be attempted for the number of times specified in the `$tries` property. After the job consumes all the retries, it will be considered as failed and will be put in the `failed_jobs` database table for inspection.

You can configure a job to retry indefinitely by providing `0` as the number of tries.

retryUntil

```
public $retryUntil = 1595049236;
```

Instead of retrying for a limited number of times, using `$retryUntil` instructs the worker to keep retrying the job until a certain time in the future.

You can add `retryUntil` as a public property on the job class or a `retryUntil` method:

```
public function retryUntil()
{
    return now()->addDay();
}
```

delay

```
public $delay = 300;
```

A `$delay` public property instructs the worker to delay processing the job until a specific number of seconds passes.

afterCommit

```
public $afterCommit = true;
```

Setting `$afterCommit = true` instructs Laravel to delay dispatching the job until any open database transactions commit. If the transaction was rolled back, the job would get discarded. And if there weren't any transactions, the job would be dispatched immediately.

uniqueId

```
public $uniqueId = 'products';
```

If a job implements the `ShouldBeUnique` or `ShouldBeUniqueUntilProcessing` interfaces. The `$uniqueId` public property will be used to create the unique key. You can also set that key by adding a `uniqueId()` method in the job class:

```
public function uniqueId()
{
    return $this->product->id;
}
```

uniqueFor

```
public $uniqueFor = 10;
```

The `$uniqueFor` property is also related to Job uniqueness. Using it you can define the number of seconds the uniqueness lock should be in place before it gets auto-released.

uniqueVia

By default, Laravel will use the default cache connection to create the atomic lock needed to ensure job uniqueness. However, you can add a `uniqueVia()` method to your job class and return a specific cache instance:

```
public function uniqueVia()
{
    return Cache::store('redis');
}
```

shouldBeEncrypted

```
public $shouldBeEncrypted = true;
```

Setting `$shouldBeEncrypted` to true instructs Laravel to store the job in the queue store in an encrypted form. Only your application workers will be able to decrypt and process encrypted jobs.

deleteWhenMissingModels

```
public $deleteWhenMissingModels = true;
```

When the `Illuminate\Queue\SerializesModels` trait is added to a job class, Laravel will handle storing pointers to eloquent models that are passed to the job constructor.

By default, if Laravel couldn't find the model while unserializing the Job payload, it's going to fail the job immediately—it will not be retried—with a `ModelNotFoundException` exception.

Using `$deleteWhenMissingModels = true` on the job class will instruct Laravel to ignore any missing models and delete the job from the queue without throwing an exception.

> **Warning:** If a job with a missing model is part of a chain, the rest of the jobs in the chain will not run.

Connection Configurations

You may configure multiple connections in the `config/queue.php` configuration file. Each queue connection uses a specific queue store, such as Redis, database, or SQS.

queue

```
'connection' => [
    'queue' => env('DEFAULT_QUEUE', 'default'),
],
```

This is the default queue jobs will be dispatched to. It's also the default queue a worker will process jobs from.

retry_after

```
'connection' => [
    'retry_after' => 90,
],
```

This value represents the number of seconds a job will remain `reserved` before it's released back to the queue. Laravel has this mechanism in place, so if a job gets stuck while processing, it gets released back automatically so another worker can pick it up.

> **Warning:** Make sure this value is greater than the timeout you set for the worker or any job-specific timeout. If you don't do that, a job may be released back to the queue while another instance of it is still being processed, which will lead to the job being processed multiple times.

block_for

```
'connection' => [
    'block_for' => 10,
],
```

This configuration is valid only for the Redis and Beanstalkd connections, and it's disabled by default. If configured, it'll keep the Redis/Beanstalkd connection open and wait for jobs to be available for the set number of seconds.

This could be useful to save time, CPU, and networking resources needed to keep pulling from the queue until a job is available. Laravel will open the connection once and wait until a job is available before it closes the connection.

after_commit

```
'connection' => [
    'after_commit' => true,
],
```

Setting `after_commit` to `true` tells Laravel to delay dispatching any job on this connection until any open database transactions commit. If the transaction was rolled back, the job would be discarded. And if there were no open transactions, the job would be dispatched right away.

Horizon Configurations

memory_limit

```
'memory_limit' => 64,
```

This configuration sets the maximum memory that may be consumed by the horizon process. This is different from the memory limit of the worker processes.

trim

```
'trim' => [
    'recent' => 10080,
    'completed' => 10080,
    'pending' => 10080,
    'recent_failed' => 10080,
    'failed' => 10080,
    'monitored' => 10080,
],
```

Horizon collects different metrics and information on jobs that are running and those that were processed. This data is stored in Redis, so it occupies valuable server memory.

Using `trim`, you can specify the number of seconds to keep every metric in memory before it gets removed automatically to free the server memory.

fast_termination

```
'fast_termination' => false,
```

By default, the horizon process, `php artisan horizon`, will wait for all workers to exit before it exits. This means Supervisor is going to wait until all existing workers exit before it

can start a new horizon process.

If there are workers in the middle of processing a long-running job, new workers will not start until these workers exit, which may cause a delay in processing new jobs.

When you set `fast_termination` to `true`, the horizon process will exit after sending the termination signal to the workers. A new horizon process can now start while the old worker processes are still finishing jobs in hand.

supervisor.balance

```
'environments' => [
    'environment' => [
        'supervisor-1' => [
            'balance' => 'simple',
        ],
    ],
],
```

This configuration key sets the balancing strategy for Horizon.

supervisor.balanceMaxShift

```
'environments' => [
    'environment' => [
        'supervisor-1' => [
            'balanceMaxShift' => 5,
        ],
    ],
],
```

This key sets the maximum number of worker processes to add or remove each time Horizon scales the workers' pool.

supervisor.balanceMaxShift

```
'environments' => [
    'environment' => [
        'supervisor-1' => [
            'balanceCooldown' => 1,
        ],
    ],
```

```
],
```

`balanceCooldown` sets the number of seconds to wait between each scaling action.

supervisor.nice

```
 'environments' => [
    'environment' => [
        'supervisor-1' => [
            'nice' => 5,
        ],
    ],
],
```

If you run your horizon workers in addition to your HTTP server on the same machine and the CPU consumption reaches 100%, the operating system will start prioritizing processes. In this scenario, you'd want to give your horizon process a lower priority so that the HTTP server can continue serving requests.

Setting a value above 0 means the process is "nice," it gives other processes more priority.

Built-in Middleware

Preventing Job Overlapping

Job concurrency is useful in many ways. In some cases, however, concurrency can lead to race conditions. For that reason, Laravel ships with a `WithoutOverlapping` middleware that prevents certain jobs from running concurrently.

This middleware was introduced in Laravel v8.11.0 (Released on October 20, 2020):

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\Middleware\WithoutOverlapping;

class UpdateBalanceJob implements ShouldQueue
{
    // The job handle method...

    public function middleware()
    {
```

```
        return [
            new WithoutOverlapping()
        ];
    }
}
```

Using the middleware inside this `UpdateBalanceJob` job will prevent workers from running multiple instances of it at the same time. If a worker picks an instance while another is processing one, it's going to release it back to the queue immediately.

Sometimes you only want to prevent overlapping of a job handling a specific resource. Let's say you want to limit updating the balance of a customer to only one instance. To do that, you can set a key:

```
public function middleware()
{
    return [
        new WithoutOverlapping($this->customer->id)
    ];
}
```

You can configure a delay for when the released instance could be attempted by using `releaseAfter`:

```
public function middleware()
{
    return [
        (new WithoutOverlapping())->releaseAfter(10)
    ];
}
```

Or you can configure the worker to delete the instance so it's not re-attempted:

```
public function middleware()
{
    return [
        (new WithoutOverlapping())->dontRelease()
    ];
}
```

This middleware uses atomic locks under the hood to prevent overlapping. The lock will be released automatically after the currently running instance runs successfully or fails. However, if the worker crashes or hangs, the lock will remain and prevent other instances from running ever. That's why it's always a good idea to set an expiration for locks:

```
public function middleware()
{
    return [
        (new WithoutOverlapping())->expireAfter(10)
    ];
}
```

Using `expireAfter()`, the lock will be released 10 seconds after the job is attempted.

Rate Limiting

Laravel also ships with a `RateLimited` job middleware. To use it, you need to define a rate limiter in one of your service providers:

```
public function boot()
{
    RateLimiter::for('reports', function ($job) {
        return $job->customer->onPremiumPlan()
                    ? Limit::perHour(100)->by($job->customer->id)
                    : Limit::perHour(10)->by($job->customer->id);
    });
}
```

Now you can use this limiter in your jobs:

```
use Illuminate\Queue\Middleware\RateLimited;

// ...

public function middleware()
{
    return [
        new RateLimited('reports')
    ];
}
```

If you're using Redis, Laravel ships another middleware that's fine-tuned for it:

```
use Illuminate\Queue\Middleware\RateLimitedWithRedis;

// ...

public function middleware()
{
    return [
        new RateLimitedWithRedis('reports')
    ];
}
```

Job Interfaces

Job Uniqueness

In the previous section, we've looked into the `WithoutOverlapping` middleware to prevent running multiple instances of the same job concurrently. However, with this middleware, multiple job instances will get dispatched to the queue. If your goal is to eventually process all these instances, then the middleware is what you're after. Otherwise, your goal may be to ensure only a single instance of a job exists in the queue at any given time. For that, we can use Laravel's job uniqueness feature.

This feature was introduced in Laravel v8.14.0 (Released on November 10, 2020). It works by implementing a `ShouldBeUnique` interface:

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateBalanceJob implements ShouldQueue, ShouldBeUnique
{

}
```

Now Laravel is going to ignore any dispatches of the `UpdateBalanceJob` job if there's already an instance of this job in the queue.

You can configure the ID to which the uniqueness will be related by adding a `$uniqueId` public property or a `uniqueId()` method:

```php
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateBalanceJob implements ShouldQueue, ShouldBeUnique
{
    public $uniqueId = 'products';

    public function uniqueId()
    {
        return $this->category->id;
    }
}
```

Under the hood, Laravel uses atomic locks to guarantee uniqueness. To configure an automatic expiration of the lock, you may add a `$uniqueFor` public property:

```php
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateBalanceJob implements ShouldQueue, ShouldBeUnique
{
    public $uniqueFor = 60;
}
```

> **Notice:** The timer starts the moment the job is dispatched to the queue, not the moment the job starts processing.

As shared earlier, Laravel will ignore all dispatches of the unique job as long as there's an instance of it still in the queue. However, sometimes you may want to start allowing dispatching the moment the job starts processing. For example, imagine a job that starts deploying a project from the latest git commit. While this job is in the queue, other deployment jobs are unnecessary since once the job in the queue runs, it's going to deploy the latest commit anyway. However, by the time this job starts running, new commits could be made. So recent dispatches of the deployment jobs are necessary to ensure these new commits are deployed.

For that reason, Laravel has another interface called `ShouldBeUniqueUntilProcessing`. When using this interface—instead of the `ShouldBeUnique` interface—Laravel will allow new dispatches the moment the job in the queue starts processing.

```
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class DeployProject implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    public $uniqueFor = 60;

    public function uniqueId()
    {
        return $this->project->id;
    }
}
```

Job Encryption

Consider this job:

```
class VerifyUser implements ShouldQueue
{
    private $user;
    private $socialSecurityNumber;

    public function __construct($user, $socialSecurityNumber)
    {
        $this->user = $user;
        $this->socialSecurityNumber = $socialSecurityNumber;
    }
}
```

When you dispatch this job, Laravel is going to serialize it so that it can be persisted in the queue store. Let's take a look at what the final form will look like in the store:

```
VerifyUser::dispatch($user, '45678AB90');
```

```
{
   "uuid":"765434b3-8251-469f-8d9b-199c89407346",
   // ...
   "data":{
      "commandName":"App\\Jobs\\VerifyUser",
"command":"O:16:\"App\\Jobs\\VerifyUser\":12:{s:22:\"\u0000App\\Jobs\\VerifyUser\u0000us
er\";N;s:38:\"\u0000App\\Jobs\\VerifyUser\u0000
```

```
socialSecurityNumber\";s:9:\"45678AB90\";s:3:\"job\";N;s:10:\"connection\";N;s:5:
\"queue\";N;s:15:\"chainConnection\";N;s:10:\"chainQueue\";N;s:19:\"chainCatchCallbacks\
";N;s:5:\"delay\";N;s:11:\"afterCommit\";N;s:10:\"middleware\";a:0:{}s:7:\"chained\";a:0
:{}}"
   }
}
```

Looking at the payload, you can see that the value of `socialSecurityNumber` is visible to the human eye. Any person—or program—that gains access to the queue store will be able to extract this value.

For most jobs, this isn't a problem. But if the job stores critical information in the payload, it's better we encrypt it so that only our queue workers can read it while processing the job. To do that, we'll need to implement the `ShouldBeEncrypted` interface:

```
use Illuminate\Contracts\Queue\ShouldBeEncrypted;

class VerifyUser implements ShouldQueue, ShouldBeEncrypted
{
    private $user;
    private $socialSecurityNumber;

    public function __construct($user, $socialSecurityNumber)
    {
        $this->user = $user;
        $this->socialSecurityNumber = $socialSecurityNumber;
    }
}
```

This interface was introduced in Laravel v8.19.0 (Released on December 15, 2020)

Now the payload will look like this:

```
{
   "uuid":"765434b3-8251-469f-8d9b-199c89407346",
   // ...
   "data":{
      "commandName":"App\\Jobs\\VerifyUser",
"command":"eyJpdiI6IjIyNWFQOXVNWn...OTJlYjBhYTFmZmQ4MjU1MDZiMDVhMjk0OTYwMTY3ZTgyYjEifQ==
"
   }
}
```

Any person or program with access to the queue store will not be able to decrypt the job payload.

You can use the `ShouldBeEncrypted` interface with queued jobs, mailables, notifications, and event listeners.