LARAVEL QUEUES IN ACTION

MOHAMED SAID

Table of Contents

Queues 101 .... **4**
Key Components 5
Queues in Laravel 8
Why Use a Message Queue? 12

Cookbook **14**
Sending Email Verification Messages 15
High Priority Jobs 22
Retrying a Verification Job 27
Canceling Abandoned Orders 32
Sending Webhooks 37
Provisioning a Forge Server 41
Canceling a Conference 47
Preventing a Double Refund 55
Batch Refunding Invoices 61
Monitoring the Refunding Process 66
Selling Conference Tickets 69
Spike Detection 73
Sending Monthly Invoices 77
Dealing With API Rate Limits 82
Limiting Job Concurrency 87
Limiting Job Rate 91
Dealing With an Unstable Service 93
That Service Is Not Even Responding 101
Refactoring to Job Middleware 104
Processing Uploaded Videos 108

Provisioning a Serverless Database 113
Generating Complex Reports 119
Message Aggregation 126
Rewarding Loyal Customers 133
Sending Deployment Notifications 138

A Guide to Running and Managing Queues **142**
Choosing the Right Machine 143
Keeping the Workers Running 149
Scaling Workers 155
Scaling With Laravel Horizon 160
Dealing With Failure 168
Choosing The Right Queue Driver 174
Dealing With Redis 180
Serverless Queues on Vapor 183
Managing Job Attempts 186
Handling Queues on Deployments 192
Designing Reliable Queued Jobs 196
The Command Bus 203

Reference **210**
Worker Configurations 211
Job Configurations 216
Connection Configurations 219
Horizon Configurations 221

Closing **224**

Queues 101

Every good technical book starts with a crash course, and this one is no exception.

In this part, we will explore the key components of a queue system, see how queues are used in Laravel, and understand why we need to use queues in our applications.

Key Components

A queue system has 3 main components; queues, messages, and workers.

Let's start by exploring each of these components.

Queues

> *A queue is a linear data structure in which elements can be added to one end, and can only be removed from the other end.*

That's the definition you find in most Computer Science books, and it can be quite confusing.

Let me show you a queue in action:

```
$queue = [
    'DownloadProject',
    'RunTests'
];
```

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

```
enqueue('Deploy');

$queue = [
    'DownloadProject',
    'RunTests',
    'Deploy'
];
```

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

```
$message = dequeue(); // == DownloadProject

$queue = [
    'RunTests',
    'Deploy'
];
```

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.

Messages

A message is a call-to-action trigger. You send a message from one part of your application and it triggers an action in another part. Or you send it from one application and it triggers an action in a completely different application.

The message sender doesn't need to worry about what happens after the message is sent, and the message receiver doesn't need to know who the sender is.

A message body contains a string, this string is interpreted by the receiver and the call to action is extracted.

Workers

A worker is a long-running process that dequeues messages and executes the call-to-action.

Here's an example worker:

```
while (true) {
    $message = dequeue();

    processMessage(
        $message
    );
}
```

Once you start a worker, it'll keep dequeuing messages from your queue. You can start as many workers as you want; the more workers you have, the faster your messages get dequeued.

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:

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

Queue::pushRaw('Send invoice #1');
```

If you're using the database queue driver, calling `pushRaw()` will add a new row in the `jobs` table with the message "Send invoice #1" stored in the `payload` field.

Enqueuing raw string messages like this is not very useful. Workers won't be able to identify the action that needs to be triggered. For that reason, Laravel allows you to enqueue class instances:

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

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

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

When you enqueue an object, the queue manager will serialize it and build a string payload for the message body. When workers dequeue that message, 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 SendInvoice
```

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

```
namespace App\Jobs;

class SendInvoice 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.

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 that I'm going to show you later.

Throughout this book, we're going to 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:

```
dispatch(
    new SendInvoice(1)
);
```

Or you can use the `Bus` facade:

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

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

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

```
SendInvoice::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 bootstrap an instance of your 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);
}
```

Re-using the same instance of your application has a major performance gain as your server will have to bootstrap the application only once during the time the worker process is alive. We'll talk more about that later.

Why Use a Message Queue?

Remember that queuing is a technique for indirect program-to-program communication. Your application can send messages to the queue, and other parts of the application can read those messages and execute them.

Here's how this can be useful:

Better User Experience

The most common reason for using a message queue is that you don't want your users to wait for certain actions to be performed before they can continue using your application.

For example, a user doesn't have to wait until your server communicates with a third-party mail service before they can complete a purchase. You can just send a success response so the user continues using the application while your server works on sending the invoice in the background.

Fault Tolerance

Queue systems are built to persist jobs until they are processed. In the case of failure, you can configure your jobs to be retried several times. If the job keeps failing, the queue manager will put it in safe storage so you can manually intervene.

For example, if your job interacts with an external service and it went down temporarily, you can configure the job to retry after some time until this service is back online.

Scalability

For applications with unpredictable load, it's a waste of money to allocate more resources all the time even when there's no load. With queues, you can control the rate at which your workers consume jobs.

Allocate more resources and your jobs will be consumed faster, limit those resources and your jobs will be consumed slower but you know they'll eventually be processed.

Batching

Batching a large task into several smaller tasks is more efficient. You can tune your queues to process these smaller tasks in parallel which will guarantee faster processing.

Ordering and Rate Limiting

With queues, you can ensure that certain jobs will be processed in a specific order. You can also limit the number of jobs running concurrently.

Cookbook

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

While doing this, I'm going to explain every queue configuration, gotcha, and technique we meet on our way.

Sending Email Verification Messages

When it comes to onboarding users to an application, providing the least amount of friction is key. However, 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 a message with a link. When the user clicks on that link, we'll know they have access to that email address.

The sign-up controller action may look like this:

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

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

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

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

After validating the user input, we add the user to the database and set `email_verified = false`. After that, we send the verification email message and redirect 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 right away and send the email message in the background.

To achieve this, we're going to configure the queue, create a job, and start a worker.

Configuring the Queue

For this challenge, we're going to use the database queue driver. It's easy to set up and gives us a high level of visibility. To begin, we need to create the migration that creates the `jobs` database table:

```
php artisan queue:table
```

Running this artisan command will create a `CreateJobsTable` migration inside the `database/migrations` directory.

Now let's run the `migrate` command:

```
php artisan migrate
```

If you check your database after this step, you 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
```

Creating and Dispatching the Job

Next, we're going to 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\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class SendVerificationMessage implements ShouldQueue
{
    use Dispatchable;
    use InteractsWithQueue;
    use Queueable;
    use 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 to the `handle()` method of the job:

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

Now we need to pass the user instance to the job constructor:

```php
public $user;

public function __construct(User $user)
{
    $this->user = $user;
}
```

To dispatch this job from the controller, we're going to use the `dispatch()` static method:

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

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

    SendVerificationMessage::dispatch($user);

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

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

When we check the `jobs` table, we are going to 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, it's a JSON string that contains information about our job and a serialized version of the `SendVerificationMessage` instance we sent.

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 you run this command, you will see information about the jobs being processed by the worker inside your 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's going to execute the `handle()` method which will run our code for sending the email message.

Running Jobs Concurrently

Right now 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 the 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 more jobs in parallel, we'll need to start more workers:

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

> **Notice:** To run multiple workers on your local machine, you need to open multiple terminal windows. One for each worker.

If we have 3 workers running and the queue is filled with jobs, we will have

3 jobs being processed at any given time.

Having more workers running speeds up the consumption of jobs and clears the queue for new jobs.

High Priority Jobs

In the previous challenge, we learned that in order 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.

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

To do that, we need to push all `ProcessPayment` jobs to a special 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 you take a look at the database queue connection in the `config/queue.php` file, you'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 same queue.

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

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

Alternatively, you 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 a queue as a `public` property, you no longer need to call `onQueue('payments')` 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, we have all our `ProcessPayment` jobs 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. These workers are processing jobs from the `default` queue by default.

Let's instruct our workers to process jobs from the `payments` queue as well. 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.

To give you an idea on how this works, let's take a look at a simplified version of the worker loop:

```
$queues = explode(',',
    $workCommand->getOption('queue')
);

while (true) {
    if ($job = getNextJob($queues)) {
        $job->run();
    }
}

function getNextJob($queues) {
    foreach($queues as $queue) {
        if ($job = $queue->pop()) {
            return $job;
        }
    }
    return null;
}
```

On each loop, the worker checks the first queue in the list—which is the `payments` queue in our example— and attempts to pop a job from it. If a job was available, it will be processed and then a new loop starts. On that new loop, the worker will also check the `payments` queue first before looking into other queues.

Jobs from the `default` queue will not be processed until the `payments` queue is cleared.

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 you want to be sure there's always a worker processing jobs from each queue, you 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 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; if the `default` queue is empty while the `payments` queue is full, the second worker will stay idle. 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. We should always try to use all available workers in the most efficient way.

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 is going to 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.

Retrying a Verification Job

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

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

If the mailing service provider had a temporary outage, an exception will be thrown and the job will fail. Laravel stores a record of failed jobs in a `failed_jobs` database table.

The record in that table may look like this:

```
{
    "id": 300,
    "uuid": "...",
    "connection": "database",
    "queue": "default",
    "payload": "{...}",
    "exception": "...",
    "failed_at": 1591425946
}
```

Retrying Manually and Automatically

To retry this job, you need to call the `queue:retry` artisan command and provide the failed job's ID:

```
php artisan queue:retry 300
```

Being able to manually retry jobs is great; but now if the job fails, the user will keep waiting for the verification email message until you manually run `queue:retry`. It would be more convenient to configure the worker to retry failed jobs a few times before it marks it as failed.

To do that, we need to start our worker with the `--tries` option:

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

This worker is going to attempt a job 3 times before it considers it a failure.

This works by dispatching the job back to the queue if it fails to complete while there are more attempts available.

Job-Specific Tries

By using the `--tries` option on the worker and setting it to attempt 3 times, all jobs consumed by this worker are going to be attempted exactly 3 times if they fail to complete.

Alternatively, we can 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;
}
```

Using the `$tries` property, we can even configure a different number of tries for each job. For example, we can configure the `ProcessPayment` job to try for 10 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 retry and only configure specific jobs to retry multiple times.

> **Notice:** A "try" translates to an attempt. Having `tries=3` means this job will be attempted 3 times, including the first run.

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.

To control the amount of time we give to our mailing service provider to recover from its temporary outage, we're going to delay the tries by setting a `$backoff` public property in the job class:

```
namespace App\Jobs;

class SendVerificationMessage implements ShouldQueue
{
    public $tries = 3;
    public $backoff = 60;
}
```

> **Notice:** In Laravel versions prior to 8.0, the `$backoff` property was called `$retryAfter` and the `--backoff` worker command option was called `--delay`.

By doing this, we're instructing our workers to retry this job after 60 seconds on each failure.

So if the job fails to complete at `22:09:00`, the workers will only try it at `22:10:00` or after. That gives the mailing provider 60 seconds—or more—to recover.

If the job was to be attempted a third time, it'll be delayed for another 60 seconds as well. So if the second attempt failed at `22:10:00`, the third attempt will occur at `22:11:00` or after.

Exponential Backoffs

If a job fails for a second time, 60 seconds might not be **not enough** time for the mailing service to recover. 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 = [60, 120];
}
```

Using an exponential backoff, we instructed the worker to delay retrying the job 60 seconds the first time, but then delay the second retry 120 seconds. The timings for the job attempts may look like this:

```
First Run At:  22:09:00
Second Run At: 22:10:00
Third Run At:  22:12:00
```

> **Notice:** If you have `$tries = 4`, the fourth attempt will be delayed 120 seconds as well.

You can also configure backoff on a worker level:

```
php artisan queue:work --backoff=60,120
```

This will delay attempting all failing jobs processed by this worker for 60 seconds after the first attempt, and 120 seconds for each attempt going forward.

Canceling Abandoned Orders

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're going to schedule a job once a user starts the checkout process. This job will check the order status after **an 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);
    }
}
```

By chaining the `delay(3600)` method after `dispatch()`, the `MonitorPendingOrder` job will be pushed to the queue with a delay of 3600 seconds (1 hour); workers will not process this job before the hour passes.

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

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

Warning: Using the SQS driver, you can only delay a job for 15 minutes. If you want to delay jobs for more, you'll need to delay for 15 minutes first and then keep releasing the job back to the queue using `release()`. You should also know that SQS stores the job for only 12 hours after it was enqueued.

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 just return from the `handle()` method. Using `return` will make the worker consider the job as successful and remove it from the queue.

Finally, we're going to cancel the order if it was still pending.

Sending Users a Reminder Before Canceling

It might be a good idea to send the user an SMS notification to remind them about their order before completely canceling it. So let's send an SMS every

15 minutes until the user completes the checkout or we cancel the order after 1 hour.

To do this, we're going to delay dispatching the job for 15 minutes instead of an hour:

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

When the job runs, we want to check if an hour has passed and cancel the order.

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

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

    if ($this->order->olderThan(59, 'minutes')) {
        $this->order->markAsCanceled();

        return;
    }

    SMS::send(...);

    $this->release(
        now()->addMinutes(15)
    );
}
```

Using `release()` inside a job has the same effect as using `delay()` while dispatching. The job will be released back to the queue and workers will run it again after 15 minutes.

Ensuring the Job Has Enough Attempts

Every time the job is released back to the queue, it'll count as an attempt. We need to make sure our job has enough `$tries` to run 4 times:

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

This job will now run:

```
15 minutes after checkout
30 minutes after checkout
45 minutes after checkout
60 minutes after checkout
```

If the user confirmed or canceled 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;
}
```

A Note on Job Delays

There's no guarantee workers will pick the job up exactly after the delay period passes. If the queue is busy and not enough workers are running, our

`MonitorPendingOrder` job may not run enough times to send the 3 SMS reminders before canceling the order.

To increase the chance of your delayed jobs getting processed on time, you need to make sure you have enough workers to empty the queue as fast as possible. This way, by the time the job becomes available, a worker process will be available to run it.

Sending Webhooks

A webhook 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::timeout(5)->post(
            $this->integration->url,
            $this->event->data
        );
    }
}
```

Notice: We configure a 5-second timeout on the client. This will prevent the job from getting stuck waiting indefinitely for a response. Always set a reasonable timeout when making requests from your code.

Handling Request Errors

If we receive a response status that's not 2xx, we want to retry the job several times before marking it as failed. But instead of retrying the job for a specific number of times, we want to retry it for a maximum of 24 hours.

First, let's catch an error response and release the job back to the queue with an exponential backoff:

```
$response = Http::timeout(...)->post(...);

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

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

What's happening in this job is that we release the job back to the queue with an increasing delay, here's how the delays will look like:

```
First run:  release(15); // 15 * 1
Second run: release(30); // 15 * 2
Third run:  release(45); // 15 * 3
```

Configuring Job Expiration

We want to keep retrying the job for a maximum of 24 hours, right? We could do the math and calculate how many attempts a full day can allow based on our exponential backoff. But there's an easier way:

```php
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.

But remember, only one attempt is allowed by default when you start a worker. In our case, we want this job to retry for an unlimited number of times since we already have an expiration in place. To do this, we need to set the `$tries` public property to `0`:

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

Now Laravel will not fail this job based on the number of attempts.

Handling Job Failure

If the HTTP request for the webhook kept failing for more than 24 hours, Laravel will mark the job as failed. In that case, we want to notify the developer of the integration so they can react.

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 is going to call this `failed()` method and pass a `MaxAttemptsExceededException` exception.

In other words, `failed()` is called when the worker is done retrying this job.

Provisioning a Forge Server

To provision a Forge server using the API, we need to send a POST request to the `/servers` endpoint. The request is going to take a few seconds and then we'll receive the server ID in the response.

Forge will continue provisioning the server for a few minutes after creating it. 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 may look:

```php
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
{
    private $server;
    private $payload;

    public function __construct(Server $server, $payload)
    {
        $this->server = $server;
        $this->payload = $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);
        }

        if ($this->server->stillProvisioning($this->server)) {
            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 yet—and create the server by contacting the Forge API and updating `forge_server_id` 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 seconds delay.

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

Managing Attempts

As we've learned by now, workers will try 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 retry 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:

```
=> Set retryUntil() to expire the job after 5 minutes.

First run happens 3 minutes from dispatch:
=> create the server
=> release(120)

The second run happens 4 minutes from the first run:
=> Job is now expired, so it's marked as failed and never runs.
```

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

To ensure this doesn't happen, we're going to rely on attempting the job for a specific number of times rather than setting a job expiration. So, if we want to give the server at least 20 minutes to provision, we'll need to attempt the job 20 times.

If the workers aren't busy, the execution times will be as follow:

```
First run  (00:00): release(120);
Second run (02:00): release(60);
Third run  (03:00): release(60);
...
20th run   (20:00): Job is marked as failed if it didn't finish.
```

Let's go ahead and set `$tries` to 20:

```
class ProvisionServer implements ShouldQueue
{
    public $tries = 20;

    // ...
}
```

Limiting Exceptions

We want the job to keep trying for at least 20 minutes, we do that by releasing the job back to the queue several times and setting the maximum number of attempts to `20`. But this also means that if an exception was thrown from inside the `handle()` method, the job will be retried 20 times.

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 20 times is too much in this case, especially if the exception needs our attention like a validation exception from Forge.

For that reason, we are going to allow the job to be released 20 times but only allow it to fail with an exception for 3 times. Here's how we'll do it:

```
class ProvisionServer implements ShouldQueue
{
    public $tries = 20;
    public $maxExceptions = 3;

    // ...
}
```

Using `$maxExceptions`, we configured the workers to fail the job if an exception was thrown from inside the `handle` method on 3 different attempts.

Now the job will be attempted 20 times, 3 out of those 20 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 exception says "Job has been attempted too many times", that means the job has exhausted all 20 attempts.

On the other hand, if the failure was due to an exception being thrown from the `handle()` method, we'll be able to see the stack trace of that exception in the `failed_jobs` table.

Canceling a Conference

It's quite unfortunate that conference organizers had to cancel every conference in 2020 due to a pandemic. But here we are facing the challenge of refunding hundreds of tickets.

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

```
class CancelConference implements ShouldQueue
{
    private $conference;

    public function __construct(Conference $conference)
    {
        $this->conference = $conference;
    }

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

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

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

Handling Timeouts

Our job may need several minutes to complete. However, workers—by default—will give every job 60 seconds to finish. We need to tell our workers to give us more time:

```
php artisan queue:work --timeout=18000
```

This worker will give every job 5 hours to report success. This might not be desired for **every** job in our application. So let's set the timeout on the job level instead:

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

Now, only the `CancelConference` job will be given 5 hours to finish while all the other jobs will be given 60 seconds.

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. Once it finishes running, it either removes the job from the queue or releases it back to be retried. When a job is released, it's no longer marked as reserved and can be picked up again by a worker.

However, if the worker process crashes while in the middle of processing the job, it will remain `reserved` forever and will never run.

To prevent 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:

```
return [

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

Since we have the timeout for this job set to 5 hours, another worker may pick it up—while still running—after 90 seconds since it'll be no longer `reserved`.

This will lead to job duplication. Very bad outcome.

To prevent this, 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' => [
            // 'driver' => 'database',
            // // ...
            'retry_after' => 18060,
        ],
    ];
];
```

Setting `retry_after` to 5 hours + 1 minute gives the worker a minute after the job timeouts to finish any housekeeping right before the 5-hour window closes.

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 won't be able to process any other jobs during that time.

When we have long-running jobs in place, it's always a good idea to make sure we have enough workers to handle other jobs. However, while having more workers, we could still get into a situation where all workers pick a 5-hour job up.

To prevent 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 the main queue.

If we dispatch our `CancelConference` job to a `cancelations` queue, assign 5 workers to that queue, and have another 5 workers processing jobs from the main queue, we'll only have 50% of our workers with a chance to be 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. That way, if the `cancellations` queue is empty the worker won't remain idle.

Using a Separate Connection

We had to configure `retry_after` a bit earlier. This will apply to all jobs dispatched to our `database` queue connection. 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,
        ],
    ];
];
```

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

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

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

```php
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.

Dispatching Multiple Jobs Instead of One

Our job may run for as long as 5 hours to refund all the attendees and send them an email one by one. 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 an exception was thrown at any point and the job was retried, we will have to start the iteration again and make sure we skip refunding invoices that were already refunded.

Instead of dispatching one long-running job, what if we dispatch multiple shorter jobs? Doing so has multiple benefits:

First, workers won't get stuck for 5 hours processing refunds for a single conference; a worker may pick up a refunding job for conference A and then pick up a refunding job for conference B on the next loop.

Second, if one refund fails, we can retry its job separately without worrying about skipping attendees who were refunded already.

Let's see how we can do that:

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

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

Here's how the job will look like:

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

    private $attendee;

    public function __construct(Attendee $attendee)
    {
        $this->attendee = $attendee;
    }

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

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

Now we can go back to using a single connection with multiple queues and have 5 workers processing jobs from the `cancelations` queue and another 5 workers processing jobs from the default queue.

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

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

Preventing a Double Refund

The `RefundAttendee` job from the previous challenge does two things; refunds the invoice and sends an email:

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

    private $attendee;

    public function __construct(Attendee $attendee)
    {
        $this->attendee = $attendee;
    }

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

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

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

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

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

Before refunding the invoice, we're going to check if it was already refunded. And after refunding, we mark it as `refunded` in our database.

That way if the job was retried, it will not have the negative side-effect of refunding the invoice multiple times.

Double Checking

When we call `refund` on an invoice, our code is going to 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 occurred 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()) {
        $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 invoice, 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:

```
Worker 1 & 2: Pick up RefundAttendee jobs for invoice #10.
Worker 1 & 2: Check wasRefunded() and both get a `false`.
Worker 1 & 2: Call refund() to refund the invoice.
```

To prevent this from happening, you can utilize atomic locks:

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

    Cache::lock('refund.'$invoice->id)
        ->get(function () use ($invoice) {
            if (! $invoice->wasRefunded()) {
                $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, we will attempt to issue the refund.

In case another worker has already acquired this lock, because it's running the duplicate job, the job will just perform nothing and return.

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

Managing Retries with Locks

If the worker crashes somewhere after acquiring the lock and before releasing it, the `RefundAttendee` job is going to be considered successful even if it never actually did the refund. That's because in future retries of the job, the lock will still be there and we won't be able to acquire a new one.

To prevent this, we need to set an 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 automatically expire the lock after 10 seconds.

Given that if the worker crashes, the job will be retried after 90 seconds, we are now worry-free as we know retries will be able to acquire a new lock.

> **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 Backoffs

If an exception was thrown, the lock will be released automatically. But what if releasing the lock failed?

The lock will automatically expire after 10 seconds, but during those 10 seconds, the job may retry and find the lock still in place and thus never completes the task.

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

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

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.

Batch Refunding Invoices

In a previous challenge we learned why it's a good idea to dispatch multiple shorter jobs instead of a long-running job:

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

Each time we call `dispatch()`, we're executing an `insert` query to store the job in our database queue driver. And if we're using Redis, it means we're executing multiple Redis commands as well.

Calling `dispatch()` multiple times means we're performing multiple round-trips to our queue store.

Dispatching a Batch

To ensure that we only execute a single store command that contains all our jobs, we may dispatch a batch:

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

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

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

Using `batch()` instructs the bus to collect and store all the jobs in a single `insert` query.

> **Notice:** Dispatching using a single command is supported in the database and Redis drivers only. Batching works in other drivers by dispatching jobs one-by-one.

Preparing for Batching

To be able to batch dispatch a job, you need to add the `Batchable` trait to it:

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

class RefundAttendee implements ShouldQueue
{
    use Batchable;
    use Dispatchable;
    use InteractsWithQueue;
    use Queueable;
    use SerializesModels;
}
```

This trait contains several methods to allow the job to interact with the batch.

Also, Laravel stores the batch information in a database table. This table doesn't exist by default. So, before you can use batches, you 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
```

Canceling a Batch

Calling `Bus::batch(...)->dispatch()` 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)->dispatch();

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

Using a reference to the `Batch`, we can allow the user to cancel the refunding process at any time.

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

$batch->cancel();
```

Calling `cancel()` will mark the batch as `canceled` in the database, but it will not prevent any jobs that are still in the queue from running.

For that reason, 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:

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

    // Actual refunding code here ...

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

Now when a worker processes a job that belongs to a canceled batch, the job will just return and the worker will consider this job as 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 Failures

With the default configurations, a batch will get canceled automatically if any of the jobs fail. That means if we're refunding 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 is going to keep the batch going without canceling it in case of failure. In addition to this, it will also store the ids of the failed jobs in the `job_batches` database table.

You can retry any of the failed jobs:

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

And you can retry all failed jobs of a given batch:

```
php artisan queue:retry-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, it's very important that we monitor the progress of the refund process.

Even though each of these `RefundAttendee` jobs is processed separately, Laravel sits there and monitors everything.

Using a reference to the batch, we can show the conference organizer information about the progress of the refund process:

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

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

Notifying the Organizer on Success

Since the refunding process can take a while, it would be nice if we inform the organizer after all attendees were refunded:

```
Bus::batch($jobs)
    ->allowFailures()
    ->then(function ($batch) {
        $conference = Conference::firstWhere(
            'refunds_batch', '=', $batch->id
        );

        Mail::to($conference->organizer)->send(
            'All attendees were refunded successfully!'
        );
    })
    ->dispatch();
```

If all the jobs in the batch finish successfully, Laravel is going to execute any callback you provide to the `then()` method. In this case, our callback is going to send an email to the organizer with a success message.

Notifying the Organizer on Failure

```
Bus::batch($jobs)
    ->allowFailures()
    ->then(...)
    ->catch(function ($batch, $e) {
        $conference = Conference::firstWhere(
            'refunds_batch', '=', $batch->id
        );

        Mail::to($conference->organizer)->send(
            'We failed to refund some of the attendees!'
        );
    })
    ->dispatch();
```

Similar to `then()`, using `catch()` allows us to execute a callback on the first failure.

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.

Notifying the Organizer on Completion

```
Bus::batch($jobs)
    ->allowFailures()
    ->then(...)
    ->catch(...)
    ->finally(function ($batch) {
        $conference = Conference::firstWhere(
            'refunds_batch', '=', $batch->id
        );

        Mail::to($conference->organizer)->send(
            'Refunding attendees completed!'
        );
    })
    ->dispatch();
```

Callbacks registered using the `finally()` method will be called after all the batch jobs are done. Some of these jobs might have completed successfully while others might have failed.

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:

```php
public function store(Conference $conference)
{
    $attendee = Attendee::create([
        'conference_id' => $conference->id
        'name' => request('name'),
        'reference' => $reference = Str::uuid()
        // ...
    ]);

    $invoice = BillingProvider::invoice([
        'customer_reference' => $reference,
        'card_token' => request('card_token'),
        // ...
    ]);

    SendTicketInformation::dispatch($attendee);
}
```

Here we create an `Attendee` model using the user information, pass the model ID as a `customer_reference` to our billing provider, and finally dispatch a `SendTicketInformation` job.

Handling Failed Payments

In case we fail to charge the user, the `BillingProvider::invoice()` method will throw an exception and the user will receive an error.

While no `SendTicketInformation` job is going to be sent in that case, we still have the attendee stored to our database. The user is now registered as an attendee!

To fix this, we could catch the exception coming from the billing service and delete the attendee record. However, a better looking alternative would be using database transactions:

```php
public function store(Conference $conference)
{
    DB::transaction(function() use ($conference) {
        $attendee = Attendee::create([
            'conference_id' => $conference->id,
            'name' => request('name'),
            'reference' => $reference = Str::uuid()
            // ...
        ]);

        $invoice = BillingProvider::invoice([
            'customer_reference' => $reference,
            'card_token' => request('card_token'),
            // ...
        ]);

        SendTicketInformation::dispatch($attendee);
    });
}
```

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

Workers Are Very Fast

If you run this code now, you may find several instances of the `SendTicketInformation` job in the `failed_jobs` table with a `ModelNotFoundException` exception.

The reason why this happened is that the job was dispatched and a worker picked it up **before** the transaction was committed.

The worker runs in a different PHP process, so when it picks the job up and tries to find that attendee in the database, it won't be able to find it because it wasn't persisted yet. This situation is called a "Race Condition".

> **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 is committed.

To prevent that race condition, we have two options; first, we can move the job dispatching part outside the transaction:

```php
$attendee = null;

DB::transaction(function() use (&$attendee, $conference) {
    $attendee = Attendee::create([
        // ...
    ]);

    // ...
});

SendTicketInformation::dispatch($attendee);
```

Now the job will be dispatched after the transaction is committed.

Another option is to dispatch the job with a delay that's long enough to guarantee the transaction is committed:

```
DB::transaction(function() use ($conference) {
    $attendee = Attendee::create([
        // ...
    ]);

    // ...

    SendTicketInformation::dispatch($attendee)->delay(5);
});
```

Now the job will not be picked up by workers before 5 seconds. This gives the transaction enough time to be committed and the attendee record to be persisted in the database.

Spike Detection

While having a spike in traffic sounds good. It's important to be alert when such a spike happens to ensure we have enough resources.

The challenge we're facing now is notifying users when there's a spike in traffic on any of their sites. The code may look like this:

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

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

Here's how the job class may look:

```
class SendSpikeDetectionNotification implements ShouldQueue
{
    // ...
    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 number of current visitors.

While the logic is valid, there is no guarantee workers will pick this job up right away. If the queue is busy, the job may get picked up after a delay. By that time, the 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 our alert 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. 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:\"Illum
inate\\Contracts\\Database\\ModelIdentifier\":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:\"ch
ainConnection\";N;s:10:\"chainQueue\";N;s:5:\"delay\";N;s:10:\"middlewar
e\";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 an `App\Site` instance.

A `ModelIdentifier` is a simple instance that holds information on the model. The worker uses this information to retrieve the model from the database when it picks the job up.

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.

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 instance of the model 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 change in state. To apply this in our challenge, let's pass the value of `current_visitors` while dispatching the job:

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

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

Our job class will now look like this:

```
class SendSpikeDetectionNotification implements ShouldQueue
{
    private $site;
    private $visitors;

    public function __construct(Conference $site, $visitors)
    {
        $this->site = $site;
        $this->visitors = $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}"
    );
}
```

Now when we send the SMS message, we're going to use the `visitors` class property instead of the `current_visitors` model attribute.

When users receive the alert, they're going to get accurate information about their site when the spike was detected.

Sending Monthly Invoices

Here's part of a challenge I've seen in a StackOverflow question:

> *At the beginning of each month, the application sends an email to every user with an invoice showing their usage throughout the month 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 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}"
    );
}
```

The currency converter sends an API request to find the current exchange rate and stores 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-local-currency 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. It'll use the rate stored inside the `$rates` property instead.

Using the Same State

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

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 you 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.

This is different from when the application is serving an HTTP request where each request is served using a dedicated process that boots up a dedicated instance.

If the currency converter was used during serving an HTTP request, it will send a request to the `ExchangeRateApi` once during the lifetime of only that request.

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 fresh exchange rate and then pass it to each job:

```php
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 fresh 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:

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

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

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

    // ...
}
```

Now the most recent exchange rate will be fetched while the jobs are being dispatched—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 as well as time. 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 start a new child process for each job in your queue.

That child job will have a fresh application instance and a fresh state.

Dealing With API Rate Limits

If your application communicates with 3rd party APIs, there's a big chance some rate limiting strategies are 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:

```php
public $tries = 10;

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

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

    // ...
}
```

If we hit a rate limit response `429 Too Many Requests`, we're going to release the job back to the queue to be retried again after 30 seconds. We also configured the job to be retried 10 times.

Not Sending Requests That We Know Will Fail

When we hit the limit, any requests sent before the limit reset point will fail. For example, if we sent all 30 requests at 10:10:45, we won't be able to send requests again before 10:11:00.

If we know requests will keep failing, there's no point in sending them and delaying processing other jobs in the queue. Instead, we're going to set a key in the cache when we hit the limit, and release the job right away if the key hasn't expired yet.

Typically when an API responds with a 429 response code, a `Retry-After` header is sent with the number of seconds to wait before 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 set an `api-limit` cache key with an expiration based on the value from the `Retry-After` header.

The value stored in the cache key will be the timestamp when requests are going to be allowed again:

```
now()->addSeconds($secondsRemaining)->timestamp
```

We're also going to use the value from `Retry-After` as a delay when releasing job:

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

That way the job is going to 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:

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

    $response = Http::acceptJson()
        ->timeout(10)
        ->withToken('...')
        ->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're only handling the 429 request error. In the actual implementation, you'll need to handle other 4xx and 5xx errors as well.

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 job was throttled by the limiter multiple times, it will not fail until the 12-hour period passes.

Limiting Exceptions

In case 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 3 attempts failed due to an exception or a timeout.

Limiting Job Concurrency

Here's a challenge I received by email from a community member.

> *A business intelligence company allowing customers to generate on-demand reports. Each customer can only have 5 reports generating at the same time.*

Here's how the job looks like:

```php
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 to only allow 5 jobs belonging to a single customer to be running at the same time. Any more report generation requests from the same customer shall be delayed until 1 of the 5 slots is freed.

Laravel ships with several built-in rate limiting helpers that we can use in our applications. In our challenge, we're going to use the concurrency limiter:

```php
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 5 concurrent executions.

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

If we were able to acquire a lock on one of the 5 slots, the limiter will execute the first closure passed to the `then()` method. 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 in order to be 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 is taking more than 60 seconds, a slot will be freed and another job may start, causing 6 concurrent jobs to be running. We don't want to allow that.

Let's configure the timeout to be 5 minutes instead:

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

Now 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` is throwing an exception, we don't want to keep retrying 10 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 10 times, 2 out of those attempts could be due to an exception being thrown or the job timing out.

Limiting Job Rate

In the previous challenge, we had to limit the concurrency of a report generation job to 5. In this example, we'll explore how we can limit generating reports to only 5 reports per hour.

Here's the `handle()` method from the previous challenge:

```php
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);
    });
}
```

Instead of using `Redis::funnel()`, we're going to use another built-in limiter called `Redis::throttle()`:

```php
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 5 reports to be generated per hour. If all slots were occupied, we're going to release the job to be retried after 10 minutes.

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

Dealing With an Unstable Service

There's this one service that keeps going down every few days. HTTP requests to their API respond with a 500 internal server error. It can take up to a couple of hours before the service goes back up.

In this challenge, we're trying to minimize the impact on our system when that service is down.

Typically we want to stop sending requests to the service if we know it has been down for a while. This will save us some valuable resources and also give a chance to other jobs in the queue—that don't communicate with this service—to be processed.

Here's how our job may look:

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

    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 certain service is failing multiple consecutive times, we're going to stop sending requests to it for a while and give it some time to recover.

To implement this pattern, we need to count the number of consecutive failures during a period of time. We're going to store this count in a cache key:

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

    if ($response->serverError()) {
        if (! Cache::get('failures')) {
            Cache::put('failures', 1, 60);
        } else {
            Cache::increment('failures');
        }

        return $this->release(600);
    }

    Cache::forget('failures');

    // Use the response to run the business logic.
}
```

Here we check if a `failures` counter doesn't exist and create it. The counter will expire and automatically be removed from the cache in 1 minute. During this time, we're going to increment the counter on each failure.

If any of the requests to the service succeeds, we're going to call `Cache::forget()` which will delete the key from the cache thus reset the counter.

Opening the Circuit

On each failure, we need to check if the number of failures crossed a certain threshold and open the circuit to stop other jobs from sending requests again for a few minutes:

```
public function handle()
{
    // ...
    //
    if ($response->serverError()) {
        // ...

        if (Cache::get('failures') > 10) {
            Cache::put('circuit:open', time(), 600);
        }

        return $this->release(600);
    }

    // ...
}
```

Here if the number of consecutive failures in a one-minute window crosses 10, we're going to store a lock in the cache that holds the current timestamp and expires after 10 minutes.

Checking for an Open Circuit

Now before running the job, we need to check if the circuit is open and release the job immediately:

```
public function handle()
{
    if ($lastFailureTimestamp = Cache::get('circuit:open')) {
        return $this->release(
            $lastFailureTimestamp + 600 + rand(1, 120)
        );
    }

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

    // ...
}
```

If the circuit is open, we're going to release the job back to the queue with a delay varying between 1 and 120 seconds after the circuit closes back again.

The reason for the random delay is to prevent flooding the service with too many requests once the circuit closes again.

> **Notice:** I studied electrical and mechanical engineering for a few years and it still confuses me a bit. An open circuit means **no flow**, a closed circuit is the opposite. (Open = don't send requests, Closed = send).

Here's the full code:

```
public function handle()
{
    if ($lastFailureTimestamp = Cache::get('circuit:open')) {
        return $this->release(
            $lastFailureTimestamp + 600 + rand(1, 120)
        );
    }

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

    if ($response->serverError()) {
        if (! $failures = Cache::get('failures')) {
            Cache::put('failures', 1, 60);
        } else {
            Cache::increment('failures');
        }

        if (Cache::get('failures') > 10) {
            Cache::put('circuit:open', time(), 600);
        }

        return $this->release(600);
    }

    Cache::forget('failures');

    // Use the response to run the business logic.
}
```

Notice: Some refactoring can be done here, but I wanted to keep the code example as clear as possible.

Now we have our circuit breaker in place, it monitors the number of failures and opens the circuit if more than 10 consecutive failures were recorded. The circuit will remain open for 10 minutes giving time for the service to recover before retrying the jobs.

If the circuit was open, jobs are going to be released back to the queue immediately with a random delay.

Checking If We Should Close the Circuit

The circuit will remain open for 10 minutes. Once it closes, we'll have to try 10 requests before our circuit breaker reaches the threshold—if the service was still down—and opens the circuit again.

It'll be a good idea to test the service with a single request before the circuit closes again. If that request succeeds, we're going to close the circuit immediately and allow new coming jobs to reach the service. If that test request fails, we're going to keep the circuit open for 10 more minutes.

```php
public function handle()
{
    if ($lastFailureTimestamp = Cache::get('circuit:open')) {
        if (time() - $lastFailureTimestamp < 8 * 60) {
            return $this->release(
                $lastFailureTimestamp + 600 + rand(1, 120)
            );
        } else {
            $halfOpen = true;
        }
    }

    // ...
}
```

If less than 8 minutes have passed since the circuit was opened, we're going to release the job to be retried later. Otherwise, we're going to let this job run even though the circuit is open. We'll call this state "half-open".

Notice: Depending on the number of workers running, multiple jobs might be considered test jobs. If you want exactly one job to run in the half-open state, you'll need to implement a lock to prevent other jobs from running.

```
if ($response->serverError()) {
    if (isset($halfOpen)) {
        Cache::put('circuit:open', time(), 600);

        return $this->release(600);
    }

    // ...
}

Cache::forget('failures');
Cache::forget('circuit:open');
```

If the test request fails, we're going to reset the timer and keep the circuit open for another 10 minutes. Otherwise, we're going to close the circuit by removing the cache key to allow jobs to run normally.

Here's the final result:

```php
public function handle()
{
    if ($lastFailureTimestamp = Cache::get('circuit:open')) {
        if (time() - $lastFailureTimestamp < 8 * 60) {
            return $this->release(
                $lastFailureTimestamp + 600 + rand(1, 120)
            );
        } else {
            $halfOpen = true;
        }
    }

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

    if ($response->serverError()) {
        if (isset($halfOpen)) {
            Cache::put('circuit:open', time(), 600);

            return $this->release(600);
        }

        if (! $failures = Cache::get('failures')) {
            Cache::put('failures', 1, 60);
        } else {
            Cache::increment('failures');
        }

        if (Cache::get('failures') > 10) {
            Cache::put('circuit:open', time(), 600);
        }

        return $this->release(600);
    }

    Cache::forget('failures');
    Cache::forget('circuit:open');

    // Use the response to run the business logic.
}
```

That Service Is Not Even Responding

That service from the previous challenge is not only responding with 500 status errors, but it also doesn't respond at all sometimes.

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

Given that we set a 10-second timeout while making requests, we're going to get a `ConnectionException` if the service doesn't respond within those 10 seconds.

To prepare for that situation, we're going to catch this exception and run our circuit breaker logic:

```
try {
    $response = Http::acceptJson()
        ->timeout(10)
        ->get('...');
} catch (ConnectionException $e) {
    // Increment the failures and do the rest of the circuit
    // breaker work.

    // Release to be retried.
}
```

Now, the circuit breaker will trip after 10 failures and this job is not going to run until the circuit closes again. During this time, workers will process other jobs as normal.

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 10 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 be waiting for the requests to timeout. No other jobs will be processed.

Bulkheading

To prevent this from happening, we can dedicate a limited number of workers to consume jobs that communicate with this service. So for example, we start 5 workers with these configurations:

```
php artisan queue:work --queues=slow,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 `slow` queue while other workers are processing other jobs.

From now on, we're going to push all the jobs communicating with the unstable service to the `slow` queue.

Limiting the effect of failure to only 5 workers is a simple form of a pattern known as Bulkheading.

Using Redis Limiters

Instead of changing your worker configurations, you could use 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 5 workers will be running this job at the same time. If another worker picks that job up while all 5 slots are occupied, it's going to release it right away giving a chance for other jobs to be processed.

Refactoring to Job Middleware

In the last few challenges, we had to write a lot of code in our job class that's not related to the job itself; we had to apply some rate-limiting as well as a circuit breaker.

In addition to adding noise around the actual job logic, this code might need to be used in several other jobs, so extracting it somewhere is a good idea.

Similar to the HTTP middleware, Laravel allows us to run our job inside a pipeline of middleware.

Here's an example middleware:

```
class RateLimitingJobMiddleware
{
    public function handle($job, $next)
    {
        Redis::throttle('job-limiter')
            ->allow(5)
            ->every(60)
            ->then(function () use ($job, $next) {
                $next($job);
            }, function () use ($job) {
                $job->release(60);
            });
    }
}
```

When our job runs through this middleware, it's going to be wrapped inside a duration rate limiter. If the limiter was able to acquire a lock, it's going to run the first callback passed to the `then()` method. Inside that callback, we have `$next($job)` which is running the `handle()` method of our job.

The `RateLimitingJobMiddleware` class can be added anywhere. I prefer creating a `App/Jobs/Middleware` directory and adding the middleware there.

Registering Middleware

To register the middleware inside your job, you need to add a `middleware` method that returns an array:

```php
public function middleware()
{
    return [new RateLimitingJobMiddleware];
}
```

While constructing the middleware object, you may pass any parameters. For example, let's pass the key to be used for the limiter.

```php
public function middleware()
{
    return [
        new RateLimitingJobMiddleware($this->customer->id)
    ];
}
```

Now inside the middleware class, we'll add a `__construct()` method:

```
class RateLimitingJobMiddleware
{
    public $key;

    public function __construct($key)
    {
        $this->key = $key;
    }

    public function handle($job, $next)
    {
        Redis::throttle(
            $this->key
        )
        // ...
    }
}
```

We can also register multiple middleware inside the `middleware` method:

```
public function middleware()
{
    return [
        new BulkHeadingJobMiddleware('slow_service'),
        new CircuitBreakerJobMiddleware('unstable_service')
    ];
}
```

Having these middleware will prevent overtaking all the workers when the service is slow using the Bulkheading pattern. It'll also open the circuit and release the job back to the queue if multiple calls to the service fail.

Implementing the Circuit Breaker in a Middleware

Here's how the handle method of the `CircuitBreakerJobMiddleware` may look:

```php
public function handle($job, $next)
{
    $this->lastFailureTimestamp = Cache::get('circuit:open');

    // Check if the circuit is open and release the job.
    if (! $this->shouldRun()) {
        return $job->release(
            $this->lastFailureTimestamp +
            $this->secondsToCloseCircuit +
            rand(1, 120)
        );
    }

    // If the circuit is closed or half-open, we will try
    // running the job and catch exceptions.
    try {
        $next($job);

        // If the job passes, we'll close the circuit if it's
        // open and reset the failures counter.
        $this->closeCircuit();
    } catch (RequestException $e) {
        if ($e->response->serverError()) {
            $this->handleFailure($job);
        }
    } catch (ConnectionException $e) {
        $this->handleFailure($job);
    }
}
```

Here we handle a server error or a `ConnectionException` and increment the failures count. Or, we trip the circuit breaker and open the circuit if the failures exceed the threshold.

Processing Uploaded Videos

Here's another challenge I found on the forums where the backend needs to receive an uploaded video from the browser, compresses that file, and then sends it to storage. The approach the developer took was:

```
class VideoUploadController{
    public function upload($request) {
        CompressAndStoreVideo::dispatch(
            $request->file('video')
        );
    }
}
```

Queueing With File Uploads

We mentioned earlier that Laravel takes a job object, serializes it, and then stores it in the queue. That means 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
```

To be able 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 file upload, 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 then start processing it.

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

Job Payload Size

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

Having a large payload means there'll be a considerable overhead each time the job is picked up by the worker. Even if the job was released back to the queue immediately due to a rate limiter, the payload will still need to be downloaded and the job will have to be loaded into your server's memory.

Moreover, you're 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 you're using the Redis queue driver, you must keep in mind that your jobs are stored in the Redis instance's memory and memory isn't unlimited. So, you need to be very careful when it comes to storing data in memory.

Reducing the Payload Size

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

The `ModelIdentifier` is a simple instance that holds the model ID, the name of the class, the name of the database connection, and names of any relationships loaded:

```php
class ModelIdentifier
{
    public function __construct(
        $class,
        $id,
        $relations,
        $connection
    )
    {
        $this->id = $id;
        $this->class = $class;
        $this->relations = $relations;
        $this->connection = $connection;
    }
}
```

Laravel uses this information to build a fresh `Model` instance when the job is being processed. This is known as Reference-Based Messaging or the Claim-Check Pattern.

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

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

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

Now inside our 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
    );
}
```

Handling Job Failure

If this job fails, Laravel is going to store it inside the `failed_jobs` database table. You can later retry this job manually or you can just report failure to the user to retry the upload.

When the job fails, the temporary file will still be occupying 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 you want to manually retry this job later, then you must keep the temporary file. In that case, you can dispatch a job to delete the temporary file—if it wasn't deleted already, after a reasonable amount of time:

```php
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 our job, we'll need to check if the temporary file still exists before running the job:

```php
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 it's not retried again.

Provisioning a Serverless Database

In this challenge, we want our users to be able to provision a serverless database on the AWS cloud platform. To do that, 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 for us 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`.

The `handle` method of our `EnsureANetworkExists` job may look like this:

```php
public function handle()
{
    $network = Network::first();

    if ($network && $network->isActive()) {
        // If we have a network and it's active already, then
        // we don't need to do anything.
        return;
    }

    if (!$network) {
        Network::create(...);
        // Create the network and check again after 2 minutes
        // to see if it became active.
        return $this->release(120);
    }

    // Check again after 2 minutes.
    return $this->release(120);
}
```

If a network doesn't exist, we create a new one and wait for it to become active. This job may be released back to the queue and attempted a few times until the network status changes to `active`.

The `EnsureNetworkHasInternetAccess` job works in a similar fashion:

```php
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:

```php
public function handle()
{
    $network = Network::first();

    if ($this->database->isActive()) {
        return;
    }

    if (! $this->database->isAttached()) {
        $network->attachDatabase(
            $this->database
        );

        return $this->release(120);
    }

    return $this->release(120);
}
```

These 3 jobs need to run one after the other. We also need to clean the resources we created in case any of the 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:

```php
$database = Database::create([
    'status' => 'provisioning'
]);

Bus::chain(
    new EnsureANetworkExists(),
    new EnsureNetworkHasInternetAccess(),
    new CreateDatabase($database)
)->dispatch();
```

First, we're going to create a database record in the database with 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. If a job didn't report completion, it'll be retried until it either succeeds and the next job in the chain is dispatched or fails and the whole chain is deleted.

Handling Failure

In case of failure, we need a way to rollback any changes that might have occurred while we're trying to create the database. In some cases that may simply 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 it's not used by any other resource. 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, that means it's safe to delete it so it doesn't cost us while not being used.

However, if it is used by other resources but they don't need internet access—no active databases—we can only remove internet access while keeping the network.

To run this compensating transaction, we can attach it to the chain using `catch()`:

```
Bus::chain(
    new EnsureANetworkExists(),
    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 we say a job *failed*, it means it has consumed all configured retries. `catch()` will not be invoked until all configured retries of a job are exhausted.

> **Notice:** The `failed()` method inside the failed job will also be triggered after the closure from the `catch()` method is invoked.

Keeping the Compensating Transaction Short

The closure passed to `catch()` will be invoked after a job exhausts all the attempts. This is done before the worker removes the job from the queue and attempts to pick up another job.

If the time the job took to run plus the time spent executing the closure from `catch()` exceeds the job timeout, the worker will exit immediately and you will end up in a state where the system is inconsistent.

So, always try to keep the tasks running inside `catch()` or inside a job's `failed()` method as short as possible. If you need to perform some lengthy work, you 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 also utilize our resources in a more efficient manner.

Given a large spreadsheet, this challenge is all 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're going to 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, we pass that model to the `ExtractData` job that's dispatched in a chain.

Once the `ExtractData` job finishes, Laravel is going to dispatch the next job in the chain. In that case, instead of a job object, we're going to dispatch a closure which will be serialized to a string format and sent to the queue.

When the closure runs, it's going to 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:

```
$jobs = $report->chunks->mapInto(
    TransformChunk::class
);

Bus::batch($jobs)->then(function() use($report) {
    Bus::chain([
        new StoreData($report),
        new GenerateSummary($report)
    ])->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:

```
START
->     RUN ExtractData
THEN
->     DISPATCH multiple TransformChunk jobs
THEN
->     RUN all TransformChunk jobs from the batch
THEN
->     DISPATCH a chain
THEN
->     RUN StoreData
THEN
->     RUN GenerateSummary
END
```

Here's the full code for the process:

```
$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();
```

Making It More Readable

The code from above can be a little confusing. And if the process is more complex, it's going to 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 `$this` inside closures. Serialization of a closure that contains `$this` is buggy.

This class represents our execution plan with its 3 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:

```php
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();
    }
}
```

Message Aggregation

Here's an interesting challenge I received via email.

An app that aggregates messages coming from different social media platforms and channels them into a single customer support platform. The challenge here is to guarantee message delivery order for each of the app customers.

Each messaging platform we integrate with sends us the messages by calling a webhook. If we dispatch a job for each message we receive, there's no guarantee our workers will pick those messages up in the correct order.

For example; while message 1 is 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 is retried.

To guarantee message delivery, we want to send message 2 **only** if message 1 was sent.

To simplify the challenge, let's assume we're only receiving messages from Telegram and sending it to Intercom. We want to make sure message 1 from Telegram to our app customer Adam is sent before message 2 to Adam. But we don't want to delay message 3 that's sent to Jane until Adam receives message 2.

Secondary Queue

To achieve this, we're going to create a secondary queue. This queue is a database table that stores all incoming messages. 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 3 to Jane",
    "customer": "Jane",
    "timestamp": "1593492166"
}

{
    "message": "Message 2 to Adam",
    "customer": "Adam",
    "timestamp": "1593492150"
}
```

To find the messages in the queue for a particular customer we can run a query like:

```
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 that's allocated to the customer's jobs only:

```
Bus::chain([
    new ProcessMessage('message 1'),
    new ProcessMessage('message 2'),
    // ...
])->dispatch();
```

To do that, we're going to push a Messenger Job once a customer account is activated. This job is going to keep an eye on the `messages` database table and pushes the jobs in the chain under its control:

```
// On customer activation

Messenger::dispatch($customer);
```

Now inside the job, we're going to run a query to collect messages from the database:

```php
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's going to collect messages for that customer and put them in a chain under the `Messenger` job.

After adding all messages to the chain, it's going to put a new instance of itself at the end of the chain so the job runs again after processing all jobs.

If no messages are available, the job is going to release itself back to the queue to run again after 5 seconds.

> **Notice:** Make sure you set `$tries = 0` so the job may be released back to the queue for an unlimited number of times. Otherwise, it will fail after a few rounds and won't process messages.

Here's how the execution plan may look:

```
START
->    RUN Messenger
->    Messenger puts 2 messages into the chain
->    Messenger puts a new instance of itself in the chain
THEN
->    RUN ProcessMessage for job 1
THEN
->    RUN ProcessMessage for job 2
THEN
->    RUN Messenger
->    ...
```

This chain is going to stay alive indefinitely looking for messages to process for this customer. Other instances of the job may also 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, 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, 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 huge payload and that 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 at all times. We want to find a way to remove this job from the queue in case we decided 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's going to report the failure and it will not be retried again.

If the customer account was activated again, we will have to dispatch a new instance of the `Messenger` job so the customer's messages get processed.

Rewarding Loyal Customers

Here's a challenge I ran into on a forum post.

In an application that's built using event-driven programming, a `NewOrderSubmitted` 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 is going to 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 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

```
use Illuminate\Contracts\Queue\ShouldQueue;

class SendInvoice implements ShouldQueue
{
    public function handle(NewOrderSubmitted $event)
    {

    }
}

class SendGiftCoupon 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 right away.

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

On 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. There's a loyalty points system 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's going to check if the customer is eligible for a reward or not.

Now, here's the real challenge; the 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 job will be processed by a worker; it might take a few milliseconds and it might take several minutes.

In addition to this, customers are not going to be rewarded on each purchase. Dispatching the job to the queue and then deciding if a gift coupon should be rewarded 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's going to call it before sending the listener to the queue. If the method returns false, the listener will not be sent to queue.

That way we check the eligibility once the order is placed which will make the decision more accurate, and also save valuable resources by not filling the queue with jobs that won't run.

Sending Deployment Notifications

Here 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:

```php
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.

Queueing Notifications

Similar to queueing listeners, we can implement the `ShouldQueue` interface in the notification class to instruct Laravel to send it to 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're sending notifications to 3 developers on the Mail and Slack channels, 6 notification jobs will be sent to the queue.

If any of the notification sending jobs fail, they can be attempted multiple times or report failure just 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 in 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.

A Guide to Running and Managing Queues

After reading through all the code challenges, I believe it's time to learn about queues!

This part is a complete guide on designing, processing, and monitoring queued jobs in a Laravel application.

Choosing the Right Machine

Using the available compute resources in the most efficient way requires multiple iterations until we find the best configuration for each use case.

Allocating less-than-needed resources to job processing and you'll start having problems with high CPU usage, jammed queues, jobs taking more time to run, timeouts, and potential crashes. Allocating more-than-needed resources and you'll be losing money.

Finding the ideal configuration for your use case is a complicated challenge that I'm going to tackle in this chapter. I want to give you a general idea of what to look for when configuring your servers to run Laravel queues.

One Dedicated vCPU

Let's start with a simple setup, a server 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 takes one or more cycles to complete.

Let's start a worker on this machine:

```
php artisan queue:work
```

A worker, under the hood, is an infinite loop that looks like this:

```php
while (true) {
    $job = $this getNextJob();

    if ($job) {
        $this->process($job);
    }
}
```

This worker process is going to use the CPU cycles available to execute the instructions needed to find the next available queued job and execute it. If no job was found, the loop will continue searching over and over.

Even though this process is an infinite loop, it won't be using all CPU cycles at all times, some of the time will be spent waiting for a response from other services. For example; if we're using the Redis queue driver, that process is going to wait for a response from the Redis instance each time it tries to fetch a new job. During this time, the processor cycles can be utilized by other processes on the machine.

> **Notice:** A process enters a waiting state due to different reasons; waiting for a response from a Redis server, a database server, an HTTP request, etc...

Now let's think about the condition where the queue isn't always filled with jobs, the process is still going to keep communicating with the queue store trying to get jobs. That continuous non-stop communication will keep the CPU busy, even though the queue is empty.

To avoid this, Laravel pauses a worker if it detects an empty queue:

```
while (true) {
    $job = $this getNextJob();

    if ($job) {
        $this->process($job);
    } else {
        sleep(3);
    }
}
```

Once the worker completes a cycle without being able to fetch any jobs, it pauses for 3 seconds. During those 3 seconds, the processor is free to execute other instructions.

We can control the number of seconds a worker pauses by providing the `--sleep` option to the worker command:

```
php artisan queue:work --sleep=1
```

Running More Workers

Now let's assume that our queues will always be full; a worker will always find jobs to run. The number of CPU cycles that worker is going to utilize depends on the kind of jobs it runs.

If the jobs mostly execute CPU intensive operations, that worker is going to utilize all the CPU cycles available preventing other processes on the server from running efficiently. However, if the jobs mostly wait, then the worker is going to spend most of its time idle.

In case the jobs are CPU intensive, we can only run a single worker on that server. That worker is going to process one job at a time, so the rate of job processing on that server depends on how fast the processor is.

On the other hand, if the jobs involve a lot of waiting, then we can run multiple workers on this single-vCPU server. While one worker is idle, another worker may use the available CPU cycles to execute jobs.

This way we are going speed up job processing by efficiently utilizing available CPU cycles.

Adding More vCPUs

Scaling the server to have more vCPUs will allow us to run more workers and thus process jobs faster. The number of workers we can run still depends on the type of jobs we are processing.

If we add too many workers, we may reach 100% CPU utilization and the server will hang. If we scale the machine to have too many vCPUs while the queue is being consumed fast then we are going to spend money for nothing.

Using a Shared vCPU

In a machine with shared vCPUs, your ability to utilize CPU cycles depends on the workload on the neighboring machines. If you share a vCPU with machines under heavy load, you will not be able to utilize the full power of the CPU. Moreover, if you push hard for too long, your server provider will send you a warning because you are trying to take all the CPU for yourself.

Shared vCPUs are a good option for low to medium workloads with occasional bursts for brief periods of time. 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.

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 that your server hits its memory limits constantly and starts swapping to disk, then you need a server with 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 that is used 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 = [];
```

If a worker process consumes all the memory allocated to PHP, the worker will exit with a Fatal error. This immediate shutdown could leave your system in a corrupted state if it happens in the middle of processing a job.

To prevent 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.

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.

Keeping the Workers Running

A worker is a PHP process that's supposed to run indefinitely. However, it may exit for several reasons:

- Reaching the memory limit
- A job timing out
- Losing connection with the database
- Process crashing

If you're using a monitoring tool, you'll be alerted when this crash happens. But it's not ideal to have to manually start workers when they crash.

Luckily there are several process managing tools we can use to start the worker processes, monitor them, and restart any crashed or exited ones.

One of the most popular tools is Supervisor, which can be easily installed on any UNIX-like operating system:

```
sudo apt-get install supervisor
sudo service supervisor restart
```

Let's configure Supervisor to start and monitor 4 worker processes:

```
[program:notification-workers]
command=php artisan queue:work notifications --tries=3 --memory=256

process_name=%(program_name)s_%(process_num)02d
autostart=true
autorestart=true
stopasgroup=true
user=forge
numprocs=4
stdout_logfile=/home/forge/.forge/notifications-workers.log
stopwaitsecs=3600
```

These configurations should be stored in a `/etc/supervisor/conf.d/notification-workers.conf` file.

Supervisor Configurations

Here's a breakdown of the configuration keys from the above example:

`command` and `numprocs`:

When Supervisor starts a program with the above configurations, it's going to create a process group with 4 processes that run the given command:

```
php artisan queue:work notifications --tries=3 --memory=256
```

`autostart`, `autorestart`, and `user`:

Each process in this group will be started when Supervisor starts and auto-restarted if it exits or crashes. The processes will be run by the `forge` system user.

`stopasgroup`:

If you explicitly instruct Supervisor to stop this group, the signal will be sent to every process in the group.

`stopwaitsecs`:

This is the amount of time Supervisor will give to a process between sending the stop signal and forcefully terminating the process.

Starting Workers

After creating this file on your system, you need to tell Supervisor about it by running the following commands:

```
supervisorctl reread
supervisorctl update

supervisorctl start notification-workers:*
```

Now Supervisor is going to start these 4 workers and keep them alive. If any of the processes crash or exit, they'll be started back.

Stopping Workers

There are cases where you need the workers to stop running; if you notice your server is running out of resources—for example—you may want to stop some of the workers to allow it to breathe. To do that, you may use the `supervisorctl stop` command:

```
supervisorctl stop notification-workers:*
```

You can also stop all workers on your machine using:

```
supervisorctl stop all
```

Using `supervisorctl stop` will send a stop signal to the workers so they finish any job currently running then stop picking up new jobs and exit.

Given the configurations from above, Supervisor will give each worker 3600 seconds to finish any job in hand and exit. You should set the value of `stopwaitsecs` to be longer than your longest running job.

That way if the worker receives a stop signal from Supervisor, it'll be able to finish running that job and exit gracefully before Supervisor terminates it.

You should also know that Supervisor will not start any workers that were stopped using `supervisorctl stop`. To start them back, you have to explicitly run `supervisorctl start`:

```
supervisorctl start notification-workers:*
```

Restarting Workers

Being a long-lived process means any changes you deploy to your code will not reflect on the instance of the application the worker is running. So after each deployment, you will need to restart all workers that run your application. To do that, you can add the `supervisorctl restart` command to your deployment script:

```
supervisorctl restart notification-workers:*
supervisorctl restart reports-workers:*
```

> **Notice:** In order to be able to run `supervisorctl` in your deployment script, you need to make sure the unix user running the deployment can run the `supervisorctl` without providing a password.

If you're using a different tool to manage your processes, you may use the generic `queue:restart` command:

```
php artisan queue:restart
```

This command will signal all workers on your machine to stop picking up new jobs and exit once they finish any job in hand.

Avoiding Memory Leaks

There has been a debate—for a long time—on whether or not PHP is a good choice for long-running processes. Based on my experience working on large scale projects that utilize hundreds of workers, I'm on the side of the debate that believes PHP is very efficient for getting the **right** job done.

The job of our workers here is running a Laravel application—PHP application—and get it to process background jobs. I've seen how powerful Laravel workers are with crunching a very large number of jobs on different server capacities and doing the job pretty well.

With that being said, avoiding memory leaks can still be a bit challenging. Over time, some references will pile up in the server memory that won't be detected by PHP and will cause the server to crash at some point.

The solution is easy though, restart the workers more often.

With a process manager like Supervisor in place, you can restart your workers every hour to clean the memory knowing that Supervisor will start them back for you automatically.

To do this, you can configure a CRON job to run every hour and call the `queue:restart` command:

```
0 * * * * forge php /home/forge/laravel.com/artisan queue:restart
```

Adding this to your `/etc/crontab` file will run `queue:restart` every hour. Workers will receive that signal and exit after finishing any job in hand.

If you don't want to use a CRON task, you can use the `--max-jobs` and `--max-time` options to limit the number of jobs the worker may process or the time it should stay up:

```
php artisan queue:work --max-jobs=1000 --max-time=3600
```

This worker will automatically exit after processing 1000 jobs or after 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.

Finally, you may signal the worker to exit from within a job `handle` method:

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

    app('queue.worker')->shouldQuit  = 1;
}
```

This method can be useful if you know this specific type of job could be memory consuming and you want to make sure the worker restarts after processing it to free any reserved resources.

Occasionally restarting your workers is a very efficient strategy for dealing with memory leaks.

Scaling Workers

For some workloads, having a fixed number of workers running at all times ensures the queue is constantly emptied and the server resources are efficiently utilized. Other workloads are variable though, in that case, we'll need to scale our workers constantly to make sure there are enough workers to process jobs at a good rate without consuming all the resources we have available.

Scaling workers can happen in the form of:

1. Adding/Removing servers to/from the cluster.
2. Starting/Stopping workers on a single server.
3. Adding/Removing resources to/from a single server.

Running on Multiple Servers

Scaling servers horizontally is managed through the server provider we're using; it'll monitor the CPU usage and memory consumption in the cluster and add or remove servers based on the scaling configurations we provide. We'll need to ensure Supervisor is installed and configured to run the required number of workers when a new server is added.

We'll also need to make sure all the servers are using the same queue connection that communicates with the same queue storage driver. For example; if we have two servers running workers that consume jobs from the database queue connection, we need to make sure both servers are connected to the same database instance.

Scaling on a Fixed Schedule

During rush hour, a carpooling application will experience a burst of traffic each workday. Similarly, a breakfast restaurant is expected to receive a lot of orders in the morning.

If we know exactly when we need to add more workers and when they are no longer needed, we can use CRON to start a supervisor process group on a fixed schedule and stop it when the workers are no longer needed.

Let's add a `/etc/supervisor/conf.d/extra-workers.conf` file:

```
[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 3 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 3 worker processes and keep them running for us.

Scaling Based on Workload

To scale workers based on workload, we'll need to monitor the number of jobs in our queue and add more workers if the queue is busy or remove workers if it's not. The number of workers to add or remove depends on the size of the server and the number of jobs in the queue.

Rather than using CRON, we're going to use the Laravel scheduler so we have access to the queue connections:

```
protected function schedule(Schedule $schedule)
{
    $schedule->call(function () {
        if (Queue::size('orders') < 30) {
            return $this->scaleDown();
        }

        Cache::increment('timer');

        if (Cache::get('timer') == 4) {
            return $this->scaleUp();
        }
    })->everyMinute();
}
```

The `scaleDown()` and `scaleUp()` methods may look like this:

```
public function scaleDown()
{
    Cache::forget('timer');

    Process::fromShellCommandline(
        'sudo supervisorctl stop extra-workers:*'
    )->run();
}

public function scaleUp()
{
    Cache::forget('timer');

    Process::fromShellCommandline(
        'sudo supervisorctl start extra-workers:*'
    )->run();
}
```

This scheduled job is added to `app/Console/Kernel.php` and runs every minute. It uses `Queue::size()` to find the number of jobs in the `orders` queue and increments a timer if that number is 30 or more.

If the value in the timer is equal to 4, that means 4 minutes have passed with the queue having at least 30 pending jobs. In that case, we start the `extra-workers:*` Supervisor group.

> **Warning:** You need to make sure you can run `supervisorctl` with `sudo` using the system user that runs the scheduler.

Using this approach, we can configure the number of workers to start (by adjusting the Supervisor configurations), how often we should run the checks, and the maximum number of jobs.

A Simpler Scaling Strategy

A simpler approach to scaling based on workload can be achieved by starting a number of workers every few minutes and instruct these workers to exit automatically when there are no more jobs to process:

```
*/5 * * * * forge php /home/forge/laravel.com/artisan
                  queue:work --stop-when-empty

*/5 * * * * forge php /home/forge/laravel.com/artisan
                  queue:work --stop-when-empty

*/5 * * * * forge php /home/forge/laravel.com/artisan
                  queue:work --stop-when-empty
```

Adding this to our `/etc/crontab` file will start 3 workers every 5 minutes. By using the `--stop-when-empty` option, those workers will exit automatically if they couldn't find any jobs to process.

However, if your queue is really busy and these workers—or some of them—didn't exit before the next set of CRON invocations, new workers may start and that can be resource consuming on your server.

If you scale your server resources automatically and you know it can handle the addition of extra workers at all times, then that's fine. However, if your resources are limited, we need to tell the workers to exit before the next invocations:

```
*/5 * * * * forge php /home/forge/laravel.com/artisan queue:work
                  --stop-when-empty
                  --max-time=240
```

Using `--max-time=240`, this worker is going to automatically exit after 4 minutes of running even if there are still jobs in the queue. That way before new extra workers are added, any extra workers still running will exit.

Scaling With Laravel Horizon

We've looked into the challenge of monitoring and scaling workers based on workload in the previous section. Using tools like Supervisor and CRON, we can scale our workers pool in and out by removing or adding workers.

> **Notice:** Scaling out means adding more workers, scaling in means removing workers. Adding/Removing workers (or servers) is called Horizontal Scaling. Increasing/Decreasing resources of a single server is called Vertical Scaling.

Designing an efficient scaling strategy can get very complicated as the application grows. Scaling multiple queues that handle thousands of jobs every hour requires constant monitoring of the system using multiple tools.

To make it easier, Laravel provides a package called Horizon that has a nice dashboard for displaying live metrics as well as a built-in auto scaler that supports different scaling strategies.

At the time of writing this book, Horizon only supports the Redis queue driver. Redis is a high-performance in-memory key-value store, which makes it a very good candidate for dealing with the monitoring challenges Horizon is built to handle.

Starting and Monitoring the Horizon Process

Instead of starting multiple workers, you only need to start the Horizon process:

```
php artisan horizon
```

You'll still want 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 to be running. No `numprocs` needs to be configured in the Supervisor configuration file.

Basic Horizon Configurations

When you install Horizon in your application, a `horizon.php` configuration file will be published in your `config` directory. At the very bottom of that file, you'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's going to 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 is going to ensure all 10 worker processes are running at all times. If one process exits, Horizon will start it back again.

You 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,
            // ...
        ],
    ],
]
```

You 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 7 worker processes that consume jobs from the `deployments` queue, 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 Scaling Strategy

Let's configure the scaling 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 5 worker processes that consume jobs from the `deployments` queue and another 5 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 3 workers for the `deployments` queue and 2 workers for the `notifications` queue.

The Auto Scaling Strategy

The most powerful scaling 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 really busy while the `notifications` queue is empty, Horizon will allocate 9 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 Scaling

By default, Horizon adds or removes a single worker every 3 seconds. That means it takes 9 seconds to scale the pool from 5 to 8 workers.

You can control the rate of scaling 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 to 3 and `balanceCooldown` equals to 1, Horizon will add or remove 3 workers every second while it scales the pool.

> **Notice:** Controlling the rate of scaling is only possible starting Horizon v5.0.

Avoiding Memory Leaks

Similar to regular worker processes, the Horizon process is a long-living PHP process. We need to make sure we free the memory used by the Horizon process as well as all the child processes running after it.

We can use the same strategy we used with regular workers, signal Horizon to terminate every hour using CRON:

```
0 * * * * forge php /home/forge/laravel.com/artisan horizon:terminate
```

When the cron runs `horizon:terminate`, Horizon will send signals to all its workers to terminate after they finish running any job in hand.

After all workers terminate, the Horizon process itself will exit. That should be enough to free the memory from any leaks coming from the worker processes or the Horizon process itself.

Dealing With Failure

A queue system involves 3 main players:

1. The producer
2. The consumer
3. The store

The producer enqueues messages, the consumer dequeues messages and processes them, and the store keeps messages until a consumer picks them up. All three players can live on the same machine or completely different machines.

Things can go wrong with any of these players or the communication channels they use. 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 in a way that keeps our system state consistent.

Failing to Serialize a Job

The queue system can only store jobs in a string form, that 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's going to throw an exception and the job will not be sent to the queue. To eliminate this kind of failure, you need to make sure your job instances along with all their dependencies are serializable.

Laravel, for example, takes care of transforming eloquent models into a serializable form when you use the `Illuminate\Queue\SerializesModels` trait in your job classes. It also wraps queued closures inside an instance of `Illuminate\Queue\SerializableClosure` which extends the opis/closure library.

It's a good idea to try and keep the properties of a queued job instance as light as possible. Instead of storing a complex object, store a reference to it, and resolve that object later when the job runs.

Failing to Send a Job

If the job was successfully serialized to a string form, the next failure point could be sending that job to the queue store. Problems may happen due to the job payload being too large 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 didn'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 2 times and sleep 5 seconds in between before throwing the exception.

Retrying immediately is helpful when the failure is temporary. Like for example, if the SQS service is down and responding with 404 errors. However, if the queue store is taking 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. Other jobs, however, are critical and they 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. 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 producer end where they keep queued messages that bounced.

```
$job = new SendOrderToSupplier($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 will start picking them up for processing. If the workers started to experience failure when trying to dequeue jobs, an exception will be reported and the worker will pause for one second before retrying again.

If you're using an error tracker like Flare or Bugsnag, you'll get alerted when the workers start throwing exceptions. Based on the situation, you may need to fix the connection issue on the queue store side or completely shut down the workers until the connection is restored so you don't waste CPU cycles.

> **Notice:** If you'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 you 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.

You can examine these failing jobs by checking the `failed_jobs` database table. You 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, you can see the exception that was thrown and caused each job to fail. However, you 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 of a job.
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, you need to make sure the `retry_after` connection configuration inside your `queue.php` configuration file is larger than the timeout of your longest-running job. That way a new job instance will not start until any running instance finishes.

Choosing The Right Queue Driver

Laravel includes built-in support for four queue storage drivers:

1. Redis
2. Database
3. SQS
4. Beanstalkd

In most cases, deciding on which queue driver to use depends on how easy you can install the driver and keep it reliably running.

The database driver is the easiest to set up and maintain since we've all been dealing with databases for some time now and feel comfortable using them. However, let's explore all the options.

The Redis Queue Driver

Redis is a high-performance in-memory key-value store, communication with it 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
```

Command execution happens, mostly, in a single thread. This means only a single command from a single client can be executed at any given time. If two workers tried to dequeue messages from a queue at the same time, there's no way both workers will end up getting the same message.

The single-threaded nature of Redis makes it a very handy tool for storing and retrieving jobs. Also since the data lives in the server memory, reading and writing operations happen very fast.

When using the Redis driver, you should know that your jobs are stored in the server memory. It's important that you ensure the payload size of your jobs is as small as possible so they don't eat the entire memory available. Also, make sure there are enough workers running to process jobs as fast as possible so they don't remain in memory for too long.

> **Notice:** A more detailed guide on dealing with Redis is included in this book.

The Database Queue Driver

We all know how to work with databases. Using a GUI tool, we can inspect the `jobs` table at any time and check all the pending jobs. The database driver gives us high visibility and it's easy to install.

In my opinion, the database driver should be the go-to driver if you are not expecting very high queue traffic to your application on day one.

Switching to a different driver is easy, so I suggest you start with the database driver until you get the notion that your queue traffic is increasing and you need a faster store.

The SQS Queue Driver

SQS is a managed queue store in the AWS cloud. If your application is hosted in AWS, communication between the application and SQS is going to be super fast. You also won't have to worry about persistence or recovery. AWS promises to handle all this for you.

However, there are several issues with SQS that you 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 your 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 2 welcome emails, for example. However, if there are certain jobs that must only run once, you need to design those jobs to be idempotent.

Idempotent jobs can run multiple times without any negative side effects. You can achieve idempotence by checking certain conditions before running each job to make sure we still need to run the job.

> **Notice:** Designing idempotent jobs is a good practice while using any queue driver, not just SQS.

Another thing with SQS is that you cannot set a `retry_after` value in your `queue.php` configuration file. You'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 either deleted or released back to the queue to be retried again.

The maximum time you can keep a job in the queue is 12 hours from the moment SQS receives the job. If you need to keep releasing the job back to the queue for more than 12 hours, you will have to delete the message and dispatch a fresh one.

The Beanstalkd Queue Driver

With the presence of Redis, database, and SQS support in Laravel, I never had to use Beanstalkd so I will skip it in this book.

The Sync Queue Driver

The `sync` driver is a special kind that stores the job in memory and processes it immediately:

```
SendInvoice::dispatch($order)->onQueue('sync');
```

In this example, you don't need to have any workers running as the `SendInvoice` job will run immediately within the existing PHP process.

This driver can be useful when you're running unit tests or while working on your application in the development/local environment.

> **Notice:** If you use the `sync` driver for development, the request object & the session will be available while the job is running. That's **not** the case while in production, you need to be aware that your jobs will run in a separate process in a completely different context.

Switching Between Drivers

To switch between queue drivers, you need to follow the following steps:

1. Configure a new queue connection in your queue.php configuration file for the new driver.
2. Update your 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 and then stop those workers.

For example, if you're switching from the `database` queue connection to the `redis` connection. You need to update your 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, you can use `Queue:size()` to find the number of pending jobs in each queue:

```
Queue::connection('database')->size('invoices')
```

When all queues are empty, you can stop the old workers to make room for adding more workers that process jobs from the new connection.

Storing Jobs On a Different Server

Having the application, workers, and queue store on the same server makes a lot of sense for a low traffic application. For high traffic ones, you may want to start using multiple servers for each service.

Using a service like Laravel Envoyer, you can deploy your application to multiple servers at the same time. You can direct your HTTP traffic to one server, use Supervisor to run your workers on another, and install and run your queue store on a third server.

This allows you to fine-tune each server separately based on what the server is supposed to do. For example, you might want to host your Redis queue store on a memory-optimized instance while running your workers on a CPU-optimized instance.

Once you configure the `queue.php` and `.env` files to use the correct credentials for your queue connection in all instances of your application, you can push jobs from one server and process them on a completely different server.

Dealing With Redis

Being an in-memory key-value store gives Redis a speed edge. However, memory isn't cheap, you need to be careful with the amount of memory your queues use so you don't end up paying too much.

Keeping Memory Usage Under Control

To keep memory usage under control, you need to work on two things:

1. Ensure the size of a job payload is as small as possible.
2. Ensure there are enough workers running to process jobs as fast as possible.

Try to keep all the job class properties as simple as possible; use strings, integers, booleans, and small value objects instead of large complex PHP objects.

If you have to pass an Eloquent Model to a job, you can pass the model ID to the job instead and retrieve the model from the database when the job runs. Or, add the `Illuminate\Queue\SerializesModels` trait to your job and Laravel will take care of that for you.

Another thing you can do is to ensure the rate of job processing is enough to free the memory allocated. If you have your memory usage constantly at 80%, for example, dispatching a large batch of jobs may cause your server to run out of memory. So always ensure your jobs are not piling up and there's enough room for new jobs to come in.

Persisting Jobs to Disk

Memory is volatile. If the server restarts while there are jobs in your queues, those jobs will be lost.

If you want, you can configure Redis to persist data on disk, either by writing to disk on fixed intervals or logging all incoming commands in an append-only fashion.

If the server is restarted, it can use the data written to disk to recover the Redis memory. You can read more on the persistence options in the Redis Guide.

Persisting to disk comes with some latency that, in my opinion, is not necessary in case of queues. If you run enough workers to ensure jobs are processed as fast as possible, you get the benefit of not worrying about losing too many jobs if the server fails, and also your application will appear fast to the end-user.

Redis Replication

Redis comes with built-in replication features. You can configure any number of read-only Redis instances to replicate data of a single leader instance.

You can use this model to push/read jobs to/from the leader instance and promote a replica to become the leader in case the leader instance crashes.

This will minimize the chances of losing jobs due to an instance that crashes. However, as mentioned in the previous section, I believe the goal should be ensuring jobs are processed as fast as possible instead of trying to focus on ways to persist or replicate queue data.

Sharing the Same Redis Instance

You can use Redis for so many applications besides queues; as a cache store or a session store for example. You can also use the same Redis instance to store jobs from multiple applications.

If you must use the same Redis instance for multiple applications, I recommend that you dedicate a single Redis database for every application. For example, you can keep database 0 for queues of application A, database 1 for the cache of application A, database 2 for queues of application B, ...

Using multiple databases allows you to run the `FLUSHDB` command on each database without affecting the data stored in other databases.

However, remember that Redis's single thread will be used to read/write data from/to all the databases. If one of the databases experience high read/write traffic, this will affect all other databases as well.

The alternative to using separate databases is running completely different redis instances. That way you can configure each instance however you see fit and ensure 100% of the Redis thread on each instance is dedicated to the application that uses it.

Serverless Queues on Vapor

Laravel Vapor is a serverless deployment platform for Laravel powered by AWS Lambda. Unlike running queues on traditional hosting services, running serverless gives you maximum scaling powers with very little overhead.

Instead of running workers, your application will receive invocations from SQS for each job you send to the queue. You can configure the number of jobs you want to process in parallel, the default timeout, and the default number of tries.

Configuring Your Queues

Vapor creates a dedicated serverless function for handling queues, you can control the maximum concurrency, timeout, and maximum memory for that function from inside the `vapor.yml` configuration file:

```
id: 1
name: laravel-queues-in-action
environments:
    production:
        queue-concurrency: 10
        queue-memory: 1024
        queue-timeout: 300
```

Having `queue-concurrency:10` means only 10 jobs will be processed in parallel at any given time. AWS will start new containers to handle your jobs up to that limit.

The Lambda-SQS integration will automatically check if there are free concurrency slots and invoke your function to process jobs based on that number.

> **Notice:** This is equivalent to running 10 workers that dequeue and run jobs from your queue.

The default concurrency limit on AWS is 1000 slots of concurrency per region. This limit includes all lambda functions your account has in a single region, which means we need to be careful while configuring that number so we give enough room for other functions to run concurrently.

Also in the example above, we have a memory limit of `1024mb` for each job processed by our lambda. Vapor will initialize the Laravel application once, per container, keep it in memory, and reuse that instance to run incoming jobs. To avoid memory leaks, Vapor will restart a container after processing 250 jobs.

> **Notice:** The minimum memory limit can be `128MB` and the maximum is `3008mb`.

Here we have also have a timeout of `300 seconds` per invocation. Any given job must run within those 5 minutes or the container will be terminated.

To prevent duplicate jobs, Vapor configures the SQS queue's `VisibilityTimeout` to have extra 10 seconds on top of the timeout. This ensures a job will not be invoked while another instance of this job is still running.

> **Notice:** The maximum timeout is `900 seconds`.

Containers and Cold Starts

When Vapor first invokes your lambda; AWS starts a fresh container, downloads your code, and initializes the layers needed to run it. This is quite some work and depending on the size of your project it could take a few seconds before your project code even runs. This is called "Cold Start".

If you have your `queue-concurrency` set to 10 and you happen to have 10 jobs ready for processing, Lambda will start 10 containers. These containers will stay alive to process more jobs up to 250 jobs.

You only pay for the number of milliseconds of work, this work could be a job being processed or the container being booted up.

Optimizing the Running Costs

If you have a queued job that only requires 128MB of memory to run, while another job requires 3GB of memory, it doesn't make sense to run the queue lambda with high memory all the time. Instead, you should consider splitting the heavy work into multiple jobs, so instead of running it inside a single lambda that requires a lot of resources, you dispatch a chain with multiple jobs and Vapor will invoke the lambda once per job.

This works in most of the cases. However, in some cases, running multiple invocations of a small lambda will cost you more than running a single invocation of one resourceful lambda. In that case, you can create a new environment with higher memory configurations that consumes jobs from a special queue and push your heavy jobs to that queue.

Remember that you pay for the execution time of your function, even if it's just sitting there and waiting for a Guzzle request to finish. There's not much you can do about it except setting a reasonable timeout for the request so you don't have to wait too long in case the external service is having networking issues.

Managing Job Attempts

When one of your workers picks up a certain job from the queue, Laravel counts it as an attempt. Even if the job didn't run at all, it's still considered an attempt. For that reason, you may want to allow your jobs to be attempted multiple times before you consider it a failure.

Let's take a look at what may cause an attempt to be considered a failure:

- Your worker fails to download the job payload due to a networking issue.
- The worker crashes while unserializing the payload.
- The job timeouts.
- An unhandled exception was thrown from inside the job.
- The worker fails to mark the job as processed due to networking issues.

The list above tells us two things; a job attempt may be considered as a failure even if the job didn't run, also an attempt can be considered as a failure even if the job did successfully run.

By default, Laravel configures a worker to attempt each job only once. This is a reasonable default for local development and staging environments, the job will fail fast so you can identify and fix the issue. However, you'll need to allow your jobs to be retried multiple times in your production environment.

You can change that default value by using the `--tries` option of the `queue:work` command:

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

You can also configure a job to be retried a certain number of times:

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

Retrying Due to a Processing Failure

If a job fails due to an exception coming from the `handle()` method of the job or a timeout, the worker will send that job back to the queue to be retried immediately.

However, sometimes you may want to delay running a retry for a few seconds/minutes. This can be done by using the `--backoff` worker option, adding a `$backoff` public property in your job class, or adding a `backoff()` method:

```
php artisan queue:work --tries=3
                       --backoff=30
```

```
class SendInvoice implements ShouldQueue{
    public $tries = 10;
    public $backoff = 30;

    // OR

    public function backoff()
    {
        return 30;
    }
}
```

You can also configure an exponential backoff to instruct the worker to delay retrying the job a different number of seconds after each attempt:

```
php artisan queue:work --tries=3
                       --backoff=30,90,300
```

```
class SendInvoice implements ShouldQueue{
    public $backoff = [30, 90, 300];

    public function backoff()
    {
        return [30, 90, 300];
    }
}
```

Here, the retry will be delayed 30 seconds the first time, then 90 seconds, then 5 minutes for each retry starting the 3rd attempt.

Retrying Due to a Worker Failure

After a failure, if the worker fails to communicate with the queue store, an attempt will be counted but the worker will not be able to configure a backoff.

In that case, the job may be retried again after the queue manager releases it back to the active queue automatically based on the `retry_after` configuration of the connection.

The timer starts when a worker picks up a job from the queue. After the configured number of seconds, the job will become available for workers to pick again for another attempt.

Warning: Make sure `retry_after` is longer than your longest-running job to prevent the job from becoming available again while another worker is still processing a previous attempt.

Retrying for a Period of Time

Sometimes you may want to keep retrying a failed job for a specific amount of time. In that case, you can use a `retryUntil()` method or a `$retryUntil` property on the job class:

```
class SendInvoice implements ShouldQueue{
    public $tries = 10;
    public $retryUntil = 1596092343;

    public function retryUntil()
    {
        return 1596092343;
    }
}
```

This job will be attempted 10 times or until Thursday, July 30, 2020, 6:59:03 AM GMT.

Notice: You can allow the job to retry for an unlimited number of times by setting `$tries = 0`.

You can also return a DateTime from within the `retryUntil` method:

```
public function retryUntil()
{
    return now()->addDays(5);
}
```

Warning: If you configure a long job expiration with an unlimited number of tries, you should consider setting a reasonable backoff so the job doesn't keep popping up in your queue and keeping it busy.

Dealing With Exceptions and timeouts

When you manually release the job back to the queue, Laravel will consider it as an attempt:

```
class SendInvoice implements ShouldQueue{
    public $tries = 10;

    public function handle()
    {
        Redis::funnel('invoices')
        ->limit(5)
        ->then(function () {
            // ...
        }, function () {
            return $this->release();
        });
    }
}
```

This job will be released back to the queue if it was throttled by the rate limiter. After 10 attempts, it'll be marked as failed.

However, you may not want this job to be attempted 10 times if there's an issue with the invoicing service that causes the job to throw an exception or timeout. In that case, you can use the `$maxExceptions` property:

```
class SendInvoice implements ShouldQueue{
    public $tries = 10;
    public $maxExceptions = 2;

    public function handle()
    {
        Redis::funnel('invoices')
        // ...
    }
}
```

This job now may be attempted for only 2 times if it fails due to an exception being thrown or a timeout. Otherwise, it'll be attempted 10 times.

> **Notice:** `$maxExceptions` must to be less than `$tries`. Unless `$tries` equals zero.

Handling Queues on Deployments

When you deploy your application with new code or different configurations, workers running on your servers need to be informed about the changes. Since workers are long-living processes, they must be shut down and restarted in order for the changes to be reflected.

Restarting Workers Through The CLI

When writing your deployment script, you need to run the following command after pulling the new changes:

```
php artisan queue:restart
```

This command will send a signal to all running workers instructing them to exit after finishing any job in hand. This is called "graceful termination".

If you're using Laravel Forge, here's a typical deployment script that you may use:

```
cd /home/forge/mysite.com
git pull origin master
$FORGE_COMPOSER install --no-interaction --prefer-dist --optimize-
autoloader

( flock -w 10 9 || exit 1
    echo 'Restarting FPM...'; sudo -S service $FORGE_PHP_FPM reload )
9>/tmp/fpmlock

$FORGE_PHP artisan migrate --force
$FORGE_PHP artisan queue:restart
```

Here the new code will be pulled from git, dependencies will be installed by composer, php-fpm will be restarted, migrations will run, and finally, the queue restart signal will be sent.

After `php-fpm` is restarted, your application visitors will start using the new code while the workers are still running on older code. Eventually, those workers will exit and be started again by Supervisor. The new worker processes will be running the new code.

If you're using Envoyer, then you need to add a deployment hook after the “Activate New Release” action and run the `queue:restart` command.

Restarting Workers Through Supervisor

If you have the worker processes managed by Supervisor, you can use the `supervisorctl` command-line tool to restart them:

```
supervisorctl restart group-name:*
```

Notice: A more detailed guide on configuring Supervisor is included.

Restarting Horizon

Similar to restarting regular worker processes, you can signal Horizon's master supervisor to terminate all worker processes by using the following command:

```
php artisan horizon:terminate
```

But in order to ensure your jobs won't be interrupted, you need to make sure of the following:

1. Your Horizon supervisors' `timeout` value is greater than the number of

seconds consumed by the longest-running job.
2. Your job-specific `timeout` is shorter than the timeout value of the Horizon supervisor.
3. If you're using the Supervisor process manager to monitor the Horizon process, make sure 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.

Horizon supervisors will also wait for the longest job to finish running and won't force-terminate after the timeout value passes.

Dealing With Migrations

When you send a restart signal to the workers, some of them may not restart right away; they'll wait for a job in hand to be processed before exiting.

If you are deploying new code along with migrations that'll change the database schema, workers that are still using the old code may fail in the middle of running their last job due to those changes; old code working with the new database schema!

To prevent this from happening, you'll need to signal the workers to exit and then wait for them. Only when all workers exit gracefully you can start your deployment.

To signal the workers to exit, you'll need to use `supervisorctl stop` in your deployment script. This command will block the execution of the script until all workers are shutdown:

```
sudo supervisorctl stop group-name:*

cd /home/forge/mysite.com
# ...

$FORGE_PHP artisan migrate --force

sudo supervisorctl start group-name:*
```

> **Warning:** Make sure the system user running the deployment can run the `supervisorctl` command as `sudo`.

Now, your workers will be signaled by Supervisor to stop after processing any jobs in hand. After all workers exit, the deployment script will continue as normal; migrations will run, and finally, the workers will be started again.

However, you should know that `supervisorctl stop` may take time to execute depending on how many workers you have and if any long-running job is being processed.

You don't want to stop the workers in this way if you don't have migrations that change the schema. So, I recommend that you don't include `supervisorctl stop` in your deployment script by default. Only include it when you know you're deploying a migration that will change the schema and cause workers running on old code to start throwing exceptions.

You can also manually run `supervisorctl stop`, wait for the command to execute, start the deployment, and finally run `supervisorctl start` after your code deploys.

Designing Reliable Queued Jobs

Queued jobs are PHP objects that get serialized, persisted, transferred, unserialized, and finally executed. They may run multiple times, on multiple 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, and it may run after a few hours.

Since the state of the system may change between the time the job was dispatched and the time it was picked up by a worker to be processed, 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 = $side;
        $this->commitHash = $commitHash;
    }
}
```

In this example job, we could have extracted the last commit hash inside the `handle` method of the job. However, by the time the job runs, new commits may have been sent to the site repository.

If the purpose of this job was to deploy the latest commit, then extracting the last commit when the job runs would have made sense. But this job deploys the last commit that was sent when the user manually triggered 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 your jobs, take a look at every piece of information the job will rely on and decide if you want this data to be independent of time or not.

Making Jobs Simple

Job objects are going to 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 that has 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.

Your job class properties should be simple datatypes; strings, integers, booleans, arrays, and the like. Or if you 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 really 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 instance of the application, resolving the cache singleton will be an easy task without much overhead.

Another thing we should do here is use the `SerializesModels` trait:

```php
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 you're using SQS, there's a payload size limit of 256 KB per job. In addition to these considerations, there's also the latency your job may add if it's too big to move fast on the network.

Try to make your job payload as light as possible. If you have to pass a big chunk of data to the job, consider storing it somewhere and pass 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.

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 opis/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 your application's key so it can validate the signature while unserializing the job to make sure 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\\Mode
lIdentifier\":4:{s:5:\"class\";s:8:\"App\\Invoice\";s:2:\"id\";N;s:9:\"r
elations\";a:0:{}s:10:\"connection\";N;}}s:8:\"
```

As you can see, Laravel takes care of converting an Eloquent model to an identifier automatically for you. But if you need to pass any complex PHP objects or a large blob of data, you should know that it'll be stored in the job payload.

If you notice your queued closure body is becoming bigger than a few lines of code, 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 in chain:

```
Bus::chain(
    new EnsureANetworkExists(),
    new EnsureNetworkHasInternetAccess(),
    new CreateDatabase()
)->dispatch();
```

Here, 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, 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.

```php
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 ran multiple times, only one refund will be sent.

Make Jobs Parallelizable

Keep in mind that multiple jobs can be running at the same time, which may lead to race conditions when they try to read/write from/to a single resource.

You may use cache locks to prevent multiple jobs from running concurrently if concurrency will have negative side effects on the overall state of your system.

You may also use funnelling to limit the number of concurrent executions of certain jobs.

The Command Bus

The command bus is the primary way of dispatching jobs to the queues in Laravel. It introduces multiple methods that allow us to dispatch a single job, a chain of jobs, or a batch of jobs. We can also use it to run a job immediately instead of sending it to the queue.

Dispatching a Job to The Queue

Using the `Bus` facade, we can dispatch a single job to the queue:

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

Bus::dispatchToQueue(
    new SendInvoice($order)
);
```

The Bus is going to use the `queue`, `connection`, and `delay` properties from the job instance to send the job to the queue. For example, if you set `$delay = 5` in your job class, the bus is going to instruct the Queue component to delay processing this job for 5 seconds.

Dispatching a Job Immediately

In some cases, you may find it more convenient to run a job immediately instead of sending it to the queue. If, for example, you're inside a queued job and want to run another job, you can run that job immediately instead of queueing it since we're already inside a background process:

```
class DeployProject implements ShouldQueue{
    public function handle()
    {
        // Deployment logic...

        Bus::dispatchNow(new RestartNginx());
    }
}
```

In the example, if the `RestartNginx` instance uses the `InteractsWithQueue` and `Queueable` traits, the Bus is going to dispatch this job to the `sync` queue driver so all the queue events are fired and the job is stored in the `failed_jobs` table if it fails. Otherwise, the `handle()` method of the `RestartNginx` job will be invoked immediately.

Dispatching a Job After Response

Normally, if you want to run a task and don't want the user to wait until it's done, you'd dispatch a queued job. However, sometimes that task is really simple and short that putting it in a queue might be an overkill.

You can run such tasks after sending the response to the user. It works by keeping the PHP process alive—to run the tasks—after closing the connection with the browser:

```
class OrderController{
    public function store()
    {
        // ...

        Bus::dispatchAfterResponse(new ReleaseLocks());

        return response('OK!');
    }
}
```

Warning: This is only a good idea if the task is really short. For long running tasks, dispatching to the queue is recommended.

Dispatching a Chain

To dispatch a chain of jobs, you may use the `chain` method:

```
Bus::chain([
    new DownloadRepo,
    new RunTests,
    new Deploy
])->dispatch();
```

Notice: Jobs in a chain run one after another. Here, the `Deploy` job will not run unless the `RunTests` job runs successfully.

You can configure the Bus to send the chain to a specific queue on a specific connection:

```
Bus::chain([
    // ...
])
->onQueue('deployments')
->onConnection('sqs')
->dispatch();
```

You may also specify a closure that'll be invoked if any of the chain jobs fail to run:

```
Bus::chain([
    // ...
])
->catch(function () {
    // Mark the deployment as failed
})
->dispatch();
```

Dispatching a Batch

To dispatch a batch of jobs, you may use the `batch` method:

```
Bus::batch([
    // ...
])
->dispatch();
```

Similar to chaining, you can also send the batch to a specific queue on a specific connection:

```
Bus::batch([
    // ...
])
->onQueue('deployments')
->onConnection('sqs')
->dispatch();
```

By default, the entire batch will be marked as failed if any of the jobs fail. You can change this behavior by using the `allowFailures()` method:

```
Bus::batch([
    // ...
])
->allowFailures()
->dispatch();
```

You can also assign multiple closures to run in different situations:

```
Bus::batch([
    // ...
])
->catch(function () {
    // A batch job has failed!
})
->then(function () {
    // All jobs have successfully run!
})
->finally(function () {
    // All jobs have run! Some may have failed.
})
->dispatch();
```

Using the dispatch() Helper

The `dispatch()` helper method can be used to dispatch a job to the queue, run it immediately, or run it after response.

```
dispatch(
    new SendInvoice($order)
);
```

Here, if the `SendInvoice` job implements the `ShouldQueue` interface, the helper will send it to the queue. Otherwise, it's going to run it immediately by invoking the handle method directly or using the `sync` queue driver.

You can also use the helper to run jobs after response:

```
dispatch(
    new SendInvoice($order)
)
->afterResponse();
```

Finally, you may set the queue, connection, or a delay:

```
dispatch(
    new SendInvoice($order)
)
->onQueue('invoices')
->onConnection('orders')
->delay(20);
```

One thing you should know about the `dispatch()` helper method is that the actual work of dispatching the job using the Bus happens when the method call goes out of context:

```
class OrderController{
    public function store()
    {
        // ...

        dispatch(new SendInvoice($order));

        $order->update([
            'status' => 'fulfilled'
        ]);

        return response('OK!');
    }
}
```

In this example, dispatching the `SendInvoice` job will happen after the `return` statement and before the response is sent. At this point, the `OrderController::store()` method's scope is terminating, and that's when the `Bus::dispatch()` method will be called.

The reason is that `dispatch()` returns an instance of `Illuminate\Foundation\Bus\PendingDispatch`, which uses the `__destruct()` magic method to execute the actual dispatching while the object is being destructed.

Here's how the `__destruct()` method looks like:

```
public function __destruct()
{
    app(\Illuminate\Contracts\Bus\Dispatcher::class)
        ->dispatch($this->job);
}
```

Using the dispatch() Static Method

The `Dispatchable` trait used in a job class contains several static methods, one of these methods is `dispatch()`:

```
SendInvoice::dispatch($order);
```

It works similar to the `dispatch()` global helper; the actual work of dispatching happens when the method call goes out of context.

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.

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 the priority of each queue based on the order in the list.

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 then 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 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
```

When your application is in maintenance mode, workers will stop picking up new jobs by default. 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 is allocated to it.

Clearing the memory before reaching the memory limit ensures your workers will 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 trying to pick up jobs again.

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 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.

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.

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 got 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.

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's occupying 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. You can find more information on the different strategies in this guide.

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 and 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 the HTTP server can continue serving requests.

Setting a value above 0 means the process is "nice," it gives other processes more priority.

Closing

The predictable and simple behavior of sequential code execution in PHP made it the most popular language in the world of backend web development. But, for many years, writing PHP programs that execute business logic in multiple processes was very difficult. Until Laravel came...

If you have used a PHP-based queue system before Laravel, you already know the value Laravel added with its queue system.

Laravel is unique in so many ways, but it's queue system makes it light-years ahead.

The simplicity of Laravel's queue system made it easier for developers to benefit from hiring workers to do the extra work in the background, while leaving the HTTP store clerks agile enough to serve more customers.

My goal from writing this book is to show you what you can do with queues, how you can avoid common issues, and how to overcome various challenges.

To me, having you say "I've seen this before!" while designing or debugging systems that use queues is the ultimate success of the book.

Rather than seeing the time spent on reading this book as school time, I hope you feel it as a chat you had with a fellow developer, who kept telling you stories about interesting problems he met, and how he managed to deal with them.

I hope you also consider telling other developers about your own stories. Write a blog post, a GitHub gist, or a tweet about the challenges you face and how you manage to deal with them by using queued jobs. Send me links—@themsaid on Twitter—and I will help spread the word as much as I can.

Finally, I'd like to thank my friends and colleagues who shared the interesting queue-related challenges they had and also proofread the book. I also want to thank my wife, not only because she supported me while working on the book, but also for editing parts of it and making it beginner friendly.

—Mohamed Said, August 2020