Course - Laravel Framework

Queue

Trong khi xây dựng ứng dụng web của mình, bạn có thể có một số tác vụ, chẳng hạn như phân tích cú pháp và lưu trữ tập tin CSV đã upload, chúng đôi khi sẽ mất rất nhiều thời gian để xử lý khi có một request. Rất may, Laravel cho phép bạn dễ dàng tạo hàng chờ cho các tác vụ này để chúng có thể được xử lý ở chế độ nền.

Tags: queue, laravel

Giới thiệu

Trong khi xây dựng ứng dụng web của mình, bạn có thể có một số tác vụ, chẳng hạn như phân tích cú pháp và lưu trữ tập tin CSV đã upload, chúng đôi khi sẽ mất rất nhiều thời gian để xử lý khi có một request. Rất may, Laravel cho phép bạn dễ dàng tạo hàng chờ cho các tác vụ này để chúng có thể được xử lý ở chế độ nền. Bằng cách đưa các tác vụ tốn nhiều thời gian vào một hàng chờ, ứng dụng của bạn có thể phản hồi các request với tốc độ nhanh hơn và cung cấp trải nghiệm người dùng tốt hơn cho khách hàng của bạn.

Hàng chờ Laravel cung cấp một API xếp hàng thống nhất dựa trên nhiều loại helper hàng chờ khác nhau, chẳng hạn như *Amazon SQS*, *Redis* hoặc thậm chí là cơ sở dữ liệu SQL.

Các tùy chọn cấu hình hàng chờ của Laravel được lưu trữ trong tập tin cấu hình *config/queue.php* của ứng dụng của bạn. Trong tập tin này, bạn sẽ tìm thấy các cấu hình kết nối cho từng driver hỗ trợ hàng chờ được đưa vào trong framework, bao gồm cơ sở dữ liệu, các driver khác như *Amazon SQS*, *Redis* và *Beanstalkd*, cũng như driver đồng bộ sẽ thực thi tác vụ ngay lập tức (để sử dụng trong quá trình phát triển trên máy cục bộ). Driver hàng chờ rỗng cũng được đưa vào nhằm mục đích loại bỏ các tác vụ đã xếp hàng chờ.

> Laravel hiện cung cấp Horizon, một dashboard và hệ thống cấu hình cho các hàng chờ được hỗ trợ bởi *Redis* của bạn. Hãy xem toàn bộ tài liệu Horizon để biết thêm thông tin.

Kết nối hay hàng chờ

Trước khi bắt đầu với hàng chờ Laravel, điều quan trọng là phải hiểu sự phân biệt giữa "kết nối" và "hàng chờ". Trong tập tin cấu hình `config/queue.php` của bạn, có một mảng cấu hình `connections`. Tùy chọn này xác định các kết nối đến các dịch vụ hàng chờ như *Amazon SQS*, *Beanstalk* hoặc *Redis*. Tuy nhiên, bất kỳ kết nối hàng chờ nào cũng có thể có nhiều "hàng chờ" có thể được coi là các ngăn xếp hoặc kho tác vụ được xếp hàng.

Lưu ý rằng mỗi ví dụ cấu hình kết nối trong tập tin cấu hình hàng chờ chứa một thuộc tính `queue`. Đây là hàng chờ mặc định mà các tác vụ sẽ được gửi đến khi chúng được gửi đến một kết nối nhất định. Nói cách khác, nếu bạn gửi một tác vụ mà không xác định rõ ràng hàng chờ nào mà nó sẽ được gửi đến, thì tác vụ sẽ được đặt trên hàng chờ được xác định trong thuộc tính `queue` của cấu hình kết nối:

```
use App\Jobs\ProcessPodcast;

// This job is sent to the default connection's default queue...
```

```
ProcessPodcast::dispatch();


// This job is sent to the default connection's "emails" queue...
ProcessPodcast::dispatch()->onQueue('emails');
```

Một số ứng dụng có thể không cần đẩy tác vụ lên nhiều hàng chờ, thay vào đó chỉ thích có một hàng chờ đơn giản. Tuy nhiên, việc đẩy tác vụ lên nhiều hàng chờ có thể đặc biệt hữu ích cho các ứng dụng muốn ưu tiên hoặc phân đoạn cách xử lý tác vụ, vì Laravel queue worker cho phép bạn chỉ định hàng chờ mà nó sẽ xử lý theo mức độ ưu tiên. Ví dụ: nếu bạn đẩy các tác vụ lên hàng đợi cao, bạn có thể chạy một worker cung cấp cho chúng mức độ ưu tiên xử lý cao hơn:

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

Ghi chú và điều kiện yêu cầu của driver

Cơ sở dữ liệu

Để sử dụng driver hàng chờ trên cơ sở dữ liệu, bạn sẽ cần một bảng cơ sở dữ liệu để chứa các tác vụ. Để tạo một migration tạo bảng này, hãy chạy lệnh Artisan `queue:table`. Khi migration đã được tạo, bạn có thể mrgrate cơ sở dữ liệu của mình bằng cách sử dụng lệnh `migrate`:

```
php artisan queue:table
```

```
php artisan migrate
```

Cuối cùng, đừng quên hướng dẫn ứng dụng của bạn sử dụng driver cơ sở dữ liệu bằng cách cập nhật biến `QUEUE_CONNECTION` trong tập tin `.env` của ứng dụng của bạn:

```
QUEUE_CONNECTION=database
```

Redis

Để sử dụng driver hàng chờ *Redis*, thì bạn nên cấu hình kết nối cơ sở dữ liệu *Redis* trong tập tin cấu hình *config/database.php* của mình.

Redis Cluster

Nếu kết nối hàng chờ *Redis* của bạn sử dụng *Redis Cluster*, thì tên hàng chờ của bạn phải chứa hashtag. Điều này là bắt buộc để đảm bảo tất cả các khóa Redis cho một hàng chờ nhất định được đặt vào cùng một vị trí hash:

```
'redis' => [
  'driver' => 'redis',
  'connection' => 'default',
  'queue' => '{default}',
  'retry_after' => 90,
],
```

Blocking

Khi sử dụng hàng chờ *Redis*, bạn có thể sử dụng tùy chọn cấu hình `block_for` để chỉ định thời gian driver sẽ đợi công việc khả dụng trước khi lặp qua vòng lặp worker và thăm dò lại cơ sở dữ liệu Redis.

Điều chỉnh giá trị này dựa trên tải trọng hàng chờ của bạn có thể hiệu quả hơn so với việc liên tục thăm dò cơ sở dữ liệu *Redis* cho các tác vụ mới. Ví dụ: bạn có thể đặt giá trị thành 5 để cho biết rằng driver sẽ block trong năm giây trong khi chờ tác vụ có sẵn:

```
'redis' => [
  'driver' => 'redis',
  'connection' => 'default',
  'queue' => 'default',
  'retry_after' => 90,
  'block_for' => 5,
],
```

> **Chú ý:** Đặt `block_for` thành `0` sẽ khiến worker hàng chờ block vô thời hạn cho đến khi có tác vụ. Điều này cũng sẽ ngăn các tín hiệu như `SIGTERM` được xử lý cho đến khi tác vụ tiếp theo được xử lý.

Các điều kiện yêu cầu về driver khác

Các package sau là cần thiết cho các driver hàng chờ tương ứng. Các package này có thể được cài đặt thông qua trình quản lý package Composer:

1. *Amazon SQS*: `aws/aws-sdk-php ~ 3.0`
2. *Beanstalkd*: `pda/pheanstalk ~ 4.0`
3. *Redis*: `predis/predis ~ 1.0` hoặc *phpredis PHP extension*

Tạo tác vụ

Tạo các class tác vụ

Theo mặc định, tất cả các tác vụ có thể xếp hàng chờ trong ứng dụng của bạn được lưu trữ trong thư mục *app/Jobs*. Nếu thư mục *app/Jobs* không tồn tại, thì nó sẽ được tạo khi bạn chạy lệnh Artisan `make:job`:

```
php artisan make:job ProcessPodcast
```

Class được tạo sẽ triển khai interface `Illuminate\Contracts\Queue\ShouldQueue`, nó cho Laravel biết rằng tác vụ nên được đẩy vào hàng chờ để chạy bất đồng bộ.

> Stub tác vụ có thể được tùy chỉnh bằng cách vận dụng stub publishing

Cấu trúc class

Các class tác vụ rất đơn giản, thường chỉ chứa một phương thức handle sẽ được gọi khi tác vụ được xử lý bởi hàng chờ. Để bắt đầu, hãy xem một class tác vụ mẫu. Trong ví dụ này, chúng ta sẽ giả sử mình đang quản lý một dịch vụ podcast và cần xử lý các tập tin podcast đã được upload trước khi chúng được xuất bản:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
```

```php
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

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

  /**
   * The podcast instance.
   *
   * @var \App\Models\Podcast
   */
  protected $podcast;

  /**
   * Create a new job instance.
   *
   * @param  App\Models\Podcast  $podcast
   * @return void
   */
  public function __construct(Podcast $podcast)
  {
    $this->podcast = $podcast;
  }

  /**
   * Execute the job.
   *
   * @param  App\Services\AudioProcessor  $processor
   * @return void
   */
  public function handle(AudioProcessor $processor)
  {
    // Process uploaded podcast...
  }
}
```

Trong ví dụ này, hãy lưu ý rằng chúng ta có thể truyền trực tiếp một mô hình Eloquent vào constructor của tác vụ sẽ được xếp hàng chờ. Do có trait `SerializesModels` mà tác vụ đang sử dụng, các mô hình Eloquent và các dữ liệu liên qua đã tải được của chúng sẽ được mã hóa và giải mã trong quá trình xử lý tác vụ.

Nếu tác vụ được xếp hàng chờ của bạn chấp nhận một mô hình Eloquent trong constructor của nó, thì chỉ identifier cho mô hình sẽ được mã hóa vào hàng chờ. Khi tác vụ thực sự được xử lý, hệ thống hàng đợi sẽ tự động truy xuất lại phiên bản mô hình đầy đủ và các dữ liệu liên qua đã tải được của nó từ cơ sở dữ liệu. Phương pháp mã hóa mô hình này cho phép gửi khối lượng tác vụ nhỏ hơn đến driver hàng chờ của bạn.

DI và `handle`

Phương thức `handle` sẽ được gọi khi tác vụ được xử lý bởi hàng chờ. Lưu ý rằng chúng tôi có thể khai báo kiểu vào phương thức `handle` của công việc. Service container của Laravel tự động đưa các package này vào.

Nếu bạn muốn toàn quyền kiểm soát cách container đưa các package này vào phương thức `handle`, thì bạn có thể sử dụng phương thức `bindMethod` của container. Phương thức `bindMethod` chấp nhận một callback nhận tác vụ và container. Trong lệnh callback, bạn có thể tự do gọi phương thức `handle` theo cách bạn muốn. Thông thường, bạn nên gọi phương thức này từ phương thức `boot` của service provider `App\Providers\AppServiceProvider`:

```
use App\Jobs\ProcessPodcast;
use App\Services\AudioProcessor;

$this->app->bindMethod([ProcessPodcast::class, 'handle'], function ($job, $app) {
  return $job->handle($app->make(AudioProcessor::class));
});
```

> **Chú ý:** Đối với dữ liệu nhị phân, chẳng hạn như nội dung hình ảnh thô, phải được truyền vào hàm `base64_encode` trước khi được chuyển đến tác vụ đã được xếp lên hàng chờ. Nếu không, tác vụ có thể không mã hóa thành JSON khi được đặt trên hàng chờ.

Các mối quan hệ được xếp hàng chờ

Bởi vì các dữ liệu liên qua đã được tải cũng được mã hóa, nên chuỗi tác vụ được mã hóa đôi

khi có thể trở nên khá lớn. Để ngăn các dữ liệu này bị mã hóa, bạn có thể gọi phương thức `withoutRelations` trên mô hình khi đặt giá trị thuộc tính. Phương thức này sẽ trả về một đối tượng của mô hình mà không tải các dữ liệu liên quan:

```
/**
 * Create a new job instance.
 *
 * @param  \App\Models\Podcast  $podcast
 * @return void
 */
public function __construct(Podcast $podcast)
{
  $this->podcast = $podcast->withoutRelations();
}
```

Hơn nữa, khi một công việc được deserialized và các mô hình liên quan được truy xuất lại từ cơ sở dữ liệu, chúng sẽ được truy xuất toàn bộ. Mọi ràng buộc về mối quan hệ trước đó được áp dụng trước khi mô hình được mã hóa trong quá trình xếp hàng tác vụ sẽ không được áp dụng khi tác vụ được deserialized. Do đó, nếu bạn muốn làm việc với một tập hợp con của một mối quan hệ nhất định, bạn nên ràng buộc lại mối quan hệ đó trong tác vụ đã được xếp hàng chờ của mình.

Các tác vụ duy nhất

> **Chú ý:** Các tác vụ duy nhất yêu cầu driver bộ nhớ cache hỗ trợ khóa. Hiện tại, driver bộ nhớ đệm `memcached`, `redis`, `dynamodb`, `database`, `file` và `array` hỗ trợ khóa. Ngoài ra, các ràng buộc tác vụ duy nhất không áp dụng cho các tác vụ trong các batch.

Đôi khi, bạn có thể muốn đảm bảo rằng chỉ một đối tượng của một tác vụ cụ thể có trong hàng chờ tại bất kỳ thời điểm nào. Bạn có thể làm như vậy bằng cách triển khai interface `ShouldBeUnique` vào class tác vụ của mình. Interface này không yêu cầu bạn khai báo bất kỳ phương thức bổ sung nào trên class của bạn:

```
<?php

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

```
class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
  ...
}
```

Trong ví dụ trên, tác vụ `UpdateSearchIndex` là duy nhất. Vì vậy, tác vụ sẽ không được gửi đi nếu một đối tượng khác của tác vụ đã có trong hàng chờ và chưa xử lý xong.

Trong một số trường hợp nhất định, bạn có thể muốn xác định một "key" cụ thể làm cho tác vụ trở thành duy nhất hoặc bạn có thể muốn chỉ định thời gian chờ mà sau đó tác vụ không còn duy nhất nữa. Để thực hiện điều này, bạn có thể xác định `uniqueId` và `uniqueF` đối với các thuộc tính hoặc phương thức trên hạng công việc của bạn:

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUnique;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
  /**
   * The product instance.
   *
   * @var \App\Product
   */
  public $product;

  /**
   * The number of seconds after which the job's unique lock will be released.
   *
   * @var int
   */
  public $uniqueFor = 3600;

  /**
   * The unique ID of the job.
```

```
     *
     * @return string
     */
    public function uniqueId()
    {
        return $this->product->id;
    }
}
```

Trong ví dụ trên, tác vụ `UpdateSearchIndex` là duy nhất bởi một ID sản phẩm. Vì vậy, bất kỳ tác vụ mới nào có cùng một ID sản phẩm sẽ bị bỏ qua cho đến khi tác vụ hiện tại hoàn tất quá trình xử lý. Ngoài ra, nếu tác vụ hiện tại không được xử lý trong một giờ, khóa duy nhất sẽ được phát hành và một tác vụ khác có cùng khóa duy nhất có thể được gửi đến hàng chờ.

Giữ tác vụ duy nhất cho đến khi bắt đầu xử lý

Theo mặc định, các tác vụ duy nhất được "unblocked" sau khi một tác vụ hoàn tất quá trình xử lý hoặc không thực hiện được tất cả các lần thử lại. Tuy nhiên, có thể có những trường hợp bạn muốn tác vụ của mình mở khóa ngay lập tức trước khi nó được xử lý. Để hoàn thành việc này, tác vụ của bạn nên triển khai contract `ShouldBeUniqueUntilProcessing` thay vì hợp đồng `ShouldBeUnique`:

```
<?php

use App\Product;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Contracts\Queue\ShouldBeUniqueUntilProcessing;

class UpdateSearchIndex implements ShouldQueue, ShouldBeUniqueUntilProcessing
{
    // ...
}
```

Khoá tác vụ duy nhất

Ở hậu trường, khi tác vụ `ShouldBeUnique` được gửi đi, Laravel cố gắng lấy một khóa

bằng khóa `uniqueId`. Nếu không có khóa, tác vụ sẽ không được gửi đi. Khóa này được giải phóng khi tác vụ hoàn thành xử lý hoặc không thực hiện được tất cả quá trình thử lại của nó. Theo mặc định, Laravel sẽ sử dụng driver bộ đệm mặc định để lấy khóa này. Tuy nhiên, nếu bạn muốn sử dụng driver khác để lấy khóa, bạn có thể khai báo phương thức `uniqueVia` sao cho trả về driver bộ đệm sẽ được sử dụng:

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

class UpdateSearchIndex implements ShouldQueue, ShouldBeUnique
{
  ...

  /**
   * Get the cache driver for the unique job lock.
   *
   * @return \Illuminate\Contracts\Cache\Repository
   */
  public function uniqueVia()
  {
    return Cache::driver('redis');
  }
}
```

> Nếu bạn chỉ cần giới hạn việc xử lý đồng thời một tác vụ, hãy sử dụng middleware tác vụ `WithoutOverlapping` để thay thế.

Middleware của tác vụ

Middleware tác vụ cho phép bạn bao bọc logic tùy chỉnh xung quanh việc thực hiện các tác vụ được xếp hàng chờ. Ví dụ: hãy xem xét phương thức `handle` sau sử dụng các tính rate limit với Redis của Laravel để chỉ cho phép một tác vụ xử lý sau mỗi năm giây:

```
use Illuminate\Support\Facades\Redis;

/**
 * Execute the job.
```

```
     *
     * @return void
     */
    public function handle()
    {
      Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
        info('Lock obtained...');

        // Handle job...
      }, function () {
        // Could not obtain lock...

        return $this->release(5);
      });
    }
```

Ngoài ra, logic rate limit này phải được sao chép cho tác vụ khác mà chúng tôi muốn rate limit.

Thay cho rate limit trong phương thức `handle`, chúng ta có thể xác định một middleware tác vụ xử lý rate limit. Laravel không có vị trí mặc định cho middleware tác vụ, vì vậy bạn có thể đặt middleware tác vụ ở bất kỳ đâu trong ứng dụng của mình. Trong ví dụ này, chúng ta sẽ đặt middleware trong thư mục *app/Jobs/Middleware*:

```
<?php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited
{
  /**
   * Process the queued job.
   *
   * @param  mixed  $job
   * @param  callable  $next
   * @return mixed
```

```
     */
    public function handle($job, $next)
    {
        Redis::throttle('key')
            ->block(0)->allow(1)->every(5)
            ->then(function () use ($job, $next) {
                // Lock obtained...

                $next($job);
            }, function () use ($job) {
                // Could not obtain lock...

                $job->release(5);
            });
    }
}
```

Như bạn có thể thấy, giống như Middleware routing, middleware tác vụ nhận tác vụ đang được xử lý và một lệnh callback sẽ được gọi để tiếp tục xử lý tác vụ.

Sau khi tạo middleware tác vụ, chúng có thể được gắn vào một tác vụ bằng cách trả lại chúng từ phương thức `middleware` của tác vụ. Phương thức này không tồn tại trên các tác vụ được tạo bởi lệnh Artisan `make:job`, vì vậy bạn sẽ cần phải thêm nó vào class tác vụ của mình theo cách thủ công:

```
use App\Jobs\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited];
}
```

> Middleware tác vụ cũng có thể được chỉ định cho chương trình theo dõi sự kiện có thể xếp hàng chờ, gửi mail và gửi thông báo.

Rate Limiting

Mặc dù chúng tôi vừa trình bày cách viết middleware cho rate limit tác vụ theo cách riêng của bạn, nhưng Laravel thực sự bao gồm một middleware rate limit mà bạn có thể vận dụng để giới hạn các tác vụ.

Ví dụ: bạn có thể muốn cho phép người dùng sao lưu dữ liệu của họ một lần mỗi giờ trong khi không áp đặt giới hạn như vậy đối với khách hàng cao cấp. Để thực hiện điều này, bạn có thể khai báo `RateLimiter` trong phương thức `boot` của `AppServiceProvider`:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;


/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  RateLimiter::for('backups', function ($job) {
    return $job->user->vipCustomer()
      ? Limit::none()
      : Limit::perHour(1)->by($job->user->id);
  });
}
```

Trong ví dụ trên, chúng tôi đã xác định giới hạn tỷ lệ theo giờ; tuy nhiên, bạn có thể dễ dàng xác định giới hạn tỷ lệ dựa trên phút bằng phương thức `perMinute`. Ngoài ra, bạn có thể truyền bất kỳ giá trị nào bạn muốn theo phương thức rate limit; tuy nhiên, giá trị này thường được sử dụng nhất để phân đoạn giới hạn tỷ lệ theo khách hàng:

```
return Limit::perMinute(50)->by($job->user->id);
```

Khi bạn đã xác định rate limit của mình, bạn có thể đính kèm rate limiter vào tác vụ sao lưu của mình bằng middleware `Illuminate\Queue\Middleware\RateLimited`. Mỗi khi công việc vượt quá rate limit, middleware này sẽ giải phóng công việc trở lại hàng chờ với một độ trễ thích hợp dựa trên thời lượng giới hạn tỷ lệ.

```
use Illuminate\Queue\Middleware\RateLimited;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [new RateLimited('backups')];
}
```

Việc giải phóng tác vụ bị rate limit trở lại hàng chờ sẽ vẫn làm tăng tổng số lần thử `attempts` của tác vụ. Bạn có thể muốn điều chỉnh các thuộc tính `tries` và `maxExceptions` trên class công việc của mình sao cho phù hợp. Hoặc, bạn có thể muốn sử dụng phương thức `retryUntil` để xác định số lượng thời gian cho đến khi công việc không còn được thử lại nữa.

Nếu bạn không muốn một công việc được thử lại khi nó bị rate limit, bạn có thể sử dụng phương thức `dontRelease`:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [(new RateLimited('backups'))->dontRelease()];
}
```

Nếu bạn đang sử dụng Redis, bạn có thể sử dụng middleware `Illuminate\Queue`

`\Middleware\RateLimitedWithRedis`, phần mềm này được tinh chỉnh cho Redis và hiệu quả hơn middleware rate limit cơ bản.

Ngăn chặn chồng chéo tác vụ

Laravel bao gồm một middleware `Illuminate\Queue\Middleware\WithoutOverlapping` cho phép bạn ngăn chồng chéo công việc dựa trên một khóa tùy ý. Điều này có thể hữu ích khi một công việc được xếp hàng đang sửa đổi một tài nguyên mà chỉ nên được sửa đổi bởi một công việc tại một thời điểm.

Ví dụ: hãy tưởng tượng bạn có một tác vụ được xếp hàng chờ cập nhật điểm tín dụng của người dùng và bạn muốn ngăn chồng chéo tác vụ cập nhật điểm tín dụng cho cùng một ID người dùng. Để thực hiện điều này, bạn có thể trả lại middleware `WithoutOverlapping` từ phương thức middleware của tác vụ của bạn:

```
use Illuminate\Queue\Middleware\WithoutOverlapping;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [new WithoutOverlapping($this->user->id)];
}
```

Mọi chồng chéo tác vụ sẽ được đưa trở lại hàng chờ. Bạn cũng có thể chỉ định số giây phải trôi qua trước khi tác vụ đã giải phóng sẽ được thử lại:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
```

```
    return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

Nếu bạn muốn xóa ngay lập tức bất kỳ chồng chéo tác vụ nào để chúng không bị thử lại, bạn có thể sử dụng phương thức `dontRelease`:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->dontRelease()];
}
```

Middleware `WithoutOverlapping` được hỗ trợ bởi tính năng khóa của Laravel. Đôi khi, công việc của bạn có thể bị lỗi hoặc hết giờ bất ngờ khiến khóa không được mở ra. Do đó, bạn có thể xác định rõ ràng thời gian hết hạn của khóa bằng cách sử dụng phương thức `expireAfter`. Ví dụ: Ví dụ dưới đây sẽ hướng dẫn Laravel giải phóng khóa `WithoutOverlapping` ba phút sau khi tác vụ bắt đầu được xử lý:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->expireAfter(180)];
}
```

> **Chú ý:** middleware `WithoutOverlapping` yêu cầu driver bộ nhớ đệm hỗ trợ khóa. Hiện tại, driver bộ nhớ đệm `memcached`, `redis`, `dynamodb`, cơ sở dữ liệu, `file` và `array` hỗ trợ khóa.

Throttling Exception

Laravel bao gồm một middleware `Illuminate\Queue\Middleware\ThrottlesExceptions` cho phép bạn điều chỉnh các exception. công việc tương tác với các dịch vụ của bên thứ ba không ổn định.

Ví dụ: hãy tưởng tượng một tác vụ được xếp hàng chờ tương tác với API của bên thứ ba bắt đầu tạo ra các exception. Để giảm bớt các exception, bạn có thể trả lại middleware `ThrottlesExceptions` từ phương thức middleware của công việc của bạn. Thông thường, middleware này phải được ghép nối với một tác vụ triển khai thời gian thử lại:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [new ThrottlesExceptions(10, 5)];
}

/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
  return now()->addMinutes(5);
}
```

Đối số đầu tiên của contructor được middleware chấp nhận là số exception mà tác vụ có thể ném ra trước khi được điều chỉnh, trong khi đối số thứ hai của contructor là số phút sẽ trôi qua trước khi tác vụ được thử lại sau khi nó đã được điều chỉnh. Trong ví dụ ở trên, nếu tác vụ có 10 exception trong vòng 5 phút, chúng tôi sẽ đợi 5 phút trước khi thử lại tác vụ.

Khi một tác vụ đưa ra exception nhưng chưa đạt đến ngưỡng exception, tác vụ thường sẽ

được thử lại ngay lập tức. Tuy nhiên, bạn có thể chỉ định số phút mà tác vụ đó sẽ bị trì hoãn bằng cách gọi phương thức `backoff` khi đính kèm middleware vào tác vụ:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [(new ThrottlesExceptions(10, 5))->backoff(5)];
}
```

Bên trong, middleware này sử dụng hệ thống bộ nhớ cache của Laravel để triển khai rate limit và tên class của tác vụ được sử dụng làm "khóa" bộ nhớ cache. Bạn có thể ghi đè khóa này bằng cách gọi theo phương thức khi đính kèm middleware vào tác vụ của mình. Điều này có thể hữu ích nếu bạn có nhiều tác vụ tương tác với cùng một dịch vụ của bên thứ ba và bạn muốn họ chia sẻ một "bucket" throttling chung:

```
use Illuminate\Queue\Middleware\ThrottlesExceptions;

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
  return [(new ThrottlesExceptions(10, 10))->by('key')];
}
```

> Nếu bạn đang sử dụng Redis, bạn có thể sử dụng middleware `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis`, được tinh chỉnh cho Redis và hiệu quả hơn middleware điều chỉnh exception cơ bản.

Điều phối tác vụ

Một khi bạn đã viết class tác vụ của mình, bạn có thể gửi nó bằng cách sử dụng phương thức `dispatch` trên chính tác vụ đó. Các đối số được truyền cho phương thức `dispatch` sẽ được cung cấp cho constructor của tác vụ:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
  /**
   * Store a new podcast.
   *
   * @param  \Illuminate\Http\Request  $request
   * @return \Illuminate\Http\Response
   */
  public function store(Request $request)
  {
    $podcast = Podcast::create(...);

    // ...

    ProcessPodcast::dispatch($podcast);
  }
}
```

Nếu bạn muốn điều động một công việc có điều kiện, bạn có thể sử dụng các phương thức `dispatchIf` và `dispatchUnless`:

```
ProcessPodcast::dispatchIf($accountActive, $podcast);
```

```
ProcessPodcast::dispatchUnless($accountSuspended, $podcast);
```

Điều phối trễ

Nếu bạn muốn chỉ định rằng một công việc không có sẵn ngay lập tức để xử lý bởi worker hàng chờ, bạn có thể sử dụng phương thức `delay` khi điều động công việc. Ví dụ: hãy xác định rằng một công việc sẽ không có sẵn để xử lý cho đến 10 phút sau khi nó đã được gửi đi:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
  /**
   * Store a new podcast.
   *
   * @param  \Illuminate\Http\Request  $request
   * @return \Illuminate\Http\Response
   */
  public function store(Request $request)
  {
    $podcast = Podcast::create(...);

    // ...

    ProcessPodcast::dispatch($podcast)
      ->delay(now()->addMinutes(10));
  }
}
```

Chú ý: Service xếp hàng chờ Amazon SQS có thời gian trễ tối đa là 15 phút.

Gửi sau khi phản hồi được gửi tới trình duyệt

Ngoài ra, phương thức `sendAfterResponse` sẽ trì hoãn việc gửi một tác vụ cho đến sau khi HTTP phản hồi được gửi đến trình duyệt của người dùng. Điều này sẽ vẫn cho phép người dùng bắt đầu sử dụng ứng dụng ngay cả khi một tác vụ được xếp hàng chờ vẫn đang thực hiện. Điều này thường chỉ được sử dụng cho các tác vụ mất khoảng một giây, chẳng

hạn như gửi email. Vì chúng được xử lý trong HTTP request hiện tại, các tác vụ được gửi theo kiểu này không yêu cầu worker hàng chờ đang chạy để chúng được xử lý:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

Bạn cũng có thể điều phối một hàm nặc danh và xâu chuỗi một phương thức `afterResponse` vào helper điều phối để thực thi hàm nặc danh sau khi phản hồi HTTP đã được gửi đến trình duyệt:

```
use App\Mail\WelcomeMessage;
use Illuminate\Support\Facades\Mail;

dispatch(function () {
  Mail::to('taylor@example.com')->send(new WelcomeMessage);
})->afterResponse();
```

Điều phối đồng bộ

Nếu bạn muốn điều phối một tác vụ ngay lập tức (đồng bộ), bạn có thể sử dụng phương thức `dispatchSync`. Khi sử dụng phương thức này, tác vụ sẽ không được xếp hàng chờ và sẽ được thực hiện ngay lập tức trong quy trình hiện tại:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
  /**
```

```
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...

        ProcessPodcast::dispatchSync($podcast);
    }
}
```

Transaction tác vụ & Cơ sở dữ liệu

Mặc dù hoàn toàn ổn khi điều phối tác vụ trong các transaction cơ sở dữ liệu, nhưng bạn nên đặc biệt lưu ý để đảm bảo rằng tác vụ của bạn sẽ thực sự có thể thực thi thành công. Khi điều động một tác vụ trong một giao dịch, có thể tác vụ đó sẽ được xử lý bởi một worker trước khi transaction đã commit. Khi điều này xảy ra, bất kỳ cập nhật nào bạn đã thực hiện cho các model hoặc record cơ sở dữ liệu trong quá trình transaction cơ sở dữ liệu có thể chưa được phản ánh trong cơ sở dữ liệu. Ngoài ra, bất kỳ model hoặc record cơ sở dữ liệu nào được tạo trong giao dịch có thể không tồn tại trong cơ sở dữ liệu.

Rất may, Laravel cung cấp một số phương pháp giải quyết vấn đề này. Trước tiên, bạn có thể đặt tùy chọn kết nối `after_commit` trong mảng cấu hình kết nối hàng chờ của mình:

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

Khi tùy chọn `after_commit` là `true`, bạn có thể điều phối tác vụ trong các transaction cơ sở dữ liệu; tuy nhiên, Laravel sẽ chờ cho đến khi tất cả các transaction cơ sở dữ liệu được mở đã được commit trước khi thực sự điều động tác vụ. Tất nhiên, nếu không có

transaction cơ sở dữ liệu nào hiện đang mở, tác vụ sẽ được điều động ngay lập tức.

Nếu một transaction bị lùi lại do một exception xảy ra trong transaction, các tác vụ đã được điều động trong transaction đó sẽ bị loại bỏ.

> Đặt tùy chọn cấu hình `after_commit` thành `true` cũng sẽ khiến mọi chương trình theo dõi event, mailables, thông báo và event truyền phát được xếp hàng chờ được gửi đi sau khi tất cả các transaction cơ sở dữ liệu mở đã được commit.

Chỉ định trực tiếp Hành vi Điều phối Commit

Nếu bạn không đặt tùy chọn cấu hình kết nối hàng chờ `after_commit` thành `true`, bạn vẫn có thể chỉ ra rằng một tác vụ cụ thể sẽ được gửi đi sau khi tất cả các transaction cơ sở dữ liệu mở đã được commit. Để thực hiện điều này, bạn có thể xâu chuỗi phương thức `afterCommit` vào hoạt động điều phối của mình:

```
use App\Jobs\ProcessPodcast;

ProcessPodcast::dispatch($podcast)->afterCommit();
```

Tương tự như vậy, nếu tùy chọn cấu hình `after_commit` được đặt thành `true`, bạn cũng vẫn có thể chỉ định cho một tác vụ cụ thể nào đó sẽ được điều động ngay lập tức mà không cần đợi bất kỳ transaction cơ sở dữ liệu mở nào commit:

```
ProcessPodcast::dispatch($podcast)->beforeCommit();
```

Chuỗi tác vụ

Chuỗi tác vụ cho phép bạn chỉ định danh sách các tác vụ được xếp hàng chờ sẽ được chạy theo trình tự sau khi tác vụ chính được thực thi thành công. Nếu một tác vụ trong chuỗi không thành công, các tác vụ còn lại sẽ không được chạy. Để thực hiện một chuỗi tác vụ được xếp hàng chờ, bạn có thể sử dụng phương thức `chain` được cung cấp bởi facade `Bus`. Lệnh `bus` của Laravel là một thành phần cấp thấp hơn mà việc điều phối tác vụ được xếp hàng chờ được xây dựng trên đó:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
```

```php
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

Bus::chain([
  new ProcessPodcast,
  new OptimizePodcast,
  new ReleasePodcast,
])->dispatch();
```

Ngoài việc xâu chuỗi các đối tượng class tác vụ, bạn cũng có thể đóng chuỗi này lại:

```php
Bus::chain([
  new ProcessPodcast,
  new OptimizePodcast,
  function () {
    Podcast::update(...);
  },
])->dispatch();
```

> **Chú ý:** Việc xóa các tác vụ bằng phương thức `$this->delete()` trong tác vụ sẽ không ngăn các tác vụ theo chuỗi được xử lý. Chuỗi sẽ chỉ ngừng thực hiện nếu một tác vụ trong chuỗi không thành công.

Kết nối trong chuỗi & hàng chờ

Nếu bạn muốn chỉ định kết nối và hàng chờ sẽ được sử dụng cho các tác vụ được xâu chuỗi, bạn có thể sử dụng các phương thức `onConnection` và `onQueue`. Các phương thức này chỉ định kết nối hàng chờ và tên hàng chờ sẽ được sử dụng trừ khi tác vụ được xếp hàng đợi được chỉ định rõ ràng một kết nối/hàng chờ khác:

```php
Bus::chain([
  new ProcessPodcast,
  new OptimizePodcast,
  new ReleasePodcast,
])->onConnection('redis')->onQueue('podcasts')->dispatch();
```

Thất bại trong xâu chuỗi

Khi xâu chuỗi các tác vụ, bạn có thể sử dụng phương thức `catch` để chỉ định một hàm nặc danh sẽ được gọi nếu một tác vụ trong chuỗi không thành công. Lệnh callback đã cho sẽ nhận được đối tượng `Throwable` đã gây ra lỗi tác vụ:

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

Bus::chain([
  new ProcessPodcast,
  new OptimizePodcast,
  new ReleasePodcast,
])->catch(function (Throwable $e) {
  // A job within the chain has failed...
})->dispatch();
```

Tùy chỉnh hàng chờ & kết nối

Điều phối một hàng chờ cụ thể

Bằng cách đẩy các tác vụ đến các hàng chờ khác nhau, bạn có thể "phân loại" các tác vụ đã xếp hàng của mình và thậm chí ưu tiên số lượng worker mà bạn chỉ định cho các hàng chờ khác nhau. Hãy nhớ rằng điều này không đẩy tác vụ đến các "kết nối" hàng chờ khác nhau như được định nghĩa bởi tập tin cấu hình hàng chờ của bạn, mà chỉ đến các hàng chờ cụ thể trong một kết nối duy nhất. Để chỉ định hàng chờ, hãy sử dụng phương thức `onQueue` khi điều động tác vụ:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;
```

```php
class PodcastController extends Controller
{
  /**
   * Store a new podcast.
   *
   * @param  \Illuminate\Http\Request  $request
   * @return \Illuminate\Http\Response
   */
  public function store(Request $request)
  {
    $podcast = Podcast::create(...);

    // Create podcast...

    ProcessPodcast::dispatch($podcast)->onQueue('processing');
  }
}
```

Ngoài ra, bạn có thể chỉ định hàng chờ của tác vụ bằng cách gọi phương thức `onQueue` trong hàm tạo của tác vụ:

```php
<?php

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 ProcessPodcast implements ShouldQueue
{
  use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

  /**
   * Create a new job instance.
```

```
     *
     * @return void
     */
    public function __construct()
    {
        $this->onQueue('processing');
    }
}
```

Điều phối một kết nối cụ thể

Nếu ứng dụng của bạn tương tác với nhiều kết nối hàng chờ, bạn có thể chỉ định kết nối nào để đẩy một tác vụ vào bằng phương thức `onConnection`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Jobs\ProcessPodcast;
use App\Models\Podcast;
use Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $podcast = Podcast::create(...);

        // Create podcast...
```

```
    ProcessPodcast::dispatch($podcast)->onConnection('sqs');
  }
}
```

Bạn có thể liên kết các phương thức `onConnection` và `onQueue` với nhau để chỉ định kết nối và hàng chờ cho một tác vụ:

```
ProcessPodcast::dispatch($podcast)
  ->onConnection('sqs')
  ->onQueue('processing');
```

Ngoài ra, bạn có thể chỉ định kết nối của tác vụ bằng cách gọi phương thức `onConnection` trong hàm tạo của công việc:

```
<?php

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 ProcessPodcast implements ShouldQueue
{
  use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

  /**
   * Create a new job instance.
   *
   * @return void
   */
  public function __construct()
  {
    $this->onConnection('sqs');
```

```
    }
}
```

Chỉ định số lần cố gắng tối đa/giá trị thời gian chờ

Số lần thử tối đa

Nếu một trong những tác vụ đã xếp hàng của bạn gặp lỗi, thì bạn có thể không muốn nó tiếp tục thử lại vô thời hạn. Do đó, Laravel cung cấp nhiều cách khác nhau để chỉ định số lần hoặc bao lâu một tác vụ có thể được thực hiện.

Một cách tiếp cận khác để chỉ định số lần tối đa mà một tác vụ có thể được thử lại đó là thông qua công tắc `--tries` trên dòng lệnh Artisan. Nó sẽ được áp dụng cho tất cả các tác vụ được xử lý bởi worker trừ khi tác vụ đang được xử lý chỉ định một số lần cụ thể mà nó có thể được thực hiện:

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

Nếu một tác vụ vượt quá số lần thử tối đa của nó, nó sẽ được coi là một tác vụ "thất bại". Để biết thêm thông tin về cách xử lý tác vụ không thành công, hãy tham khảo tài liệu tác vụ không thành công.

Bạn có thể thực hiện một cách tiếp cận chi tiết hơn bằng cách xác định số lần tối đa một tác vụ có thể được thực hiện trên chính loại tác vụ đó. Nếu số lần thử tối đa được chỉ định cho tác vụ, nó sẽ được ưu tiên hơn giá trị `--tries` được cung cấp trên dòng lệnh:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;
```

```
}
```

Thử lại dựa trên thời gian

Để thay thế cho việc xác định số lần một tác vụ có thể được thử trước khi nó thất bại, bạn có thể xác định thời điểm mà tác vụ không nên được thực hiện nữa. Điều này cho phép một tác vụ được thực hiện bất kỳ số lần nào trong một khung thời gian nhất định. Để xác định thời gian mà một tác vụ không nên được thực hiện nữa, hãy thêm phương thức `retryUntil` vào class tác vụ của bạn. Phương thức này sẽ trả về một đối tượng `DateTime`:

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
  return now()->addMinutes(10);
}
```

> Bạn cũng có thể khai báo thuộc tính `tries` hoặc phương thức `retryUntil` trên các chương trình theo dõi event đã được xếp hàng chờ của bạn.

Các exception tối đa

Đôi khi bạn có thể muốn chỉ định rằng một tác vụ có thể được thử nhiều lần, nhưng sẽ không thành công nếu các lần thử lại được kích hoạt bởi một số trường hợp ngoại lệ chưa được xử lý nào đó (trái ngược với việc được thực hiện bằng phương thức `release` một cách trực tiếp). Để thực hiện điều này, bạn có thể xác định thuộc tính `maxExceptions` trên class tác vụ của mình:

```
<?php

namespace App\Jobs;
```

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

class ProcessPodcast implements ShouldQueue
{
  /**
   * The number of times the job may be attempted.
   *
   * @var int
   */
  public $tries = 25;

  /**
   * The maximum number of unhandled exceptions to allow before failing.
   *
   * @var int
   */
  public $maxExceptions = 3;

  /**
   * Execute the job.
   *
   * @return void
   */
  public function handle()
  {
    Redis::throttle('key')->allow(10)->every(60)->then(function () {
      // Lock obtained, process the podcast...
    }, function () {
      // Unable to obtain lock...
      return $this->release(10);
    });
  }
}
```

Trong ví dụ này, tác vụ được giải phóng trong mười giây nếu ứng dụng không thể lấy được *Redis lock* và sẽ tiếp tục được thử lại tối đa 25 lần. Tuy nhiên, tác vụ sẽ thất bại nếu tác vụ ném ra ba trường hợp ngoại lệ chưa được khắc phục.

Timeout

> **Chú ý:** Phần mở rộng PHP `pcntl` phải được cài đặt để chỉ định thời gian chờ công việc.

Thông thường, bạn sẽ biết đại khái là bạn mong đợi các tác vụ xếp hàng chờ của mình mất bao lâu. Vì lý do này, Laravel cho phép bạn chỉ định giá trị "thời gian chờ". Nếu một tác vụ đang xử lý lâu hơn số giây được chỉ định bởi giá trị thời gian chờ, worker đang xử lý công việc sẽ thoát ra với một lỗi. Thông thường, công nhân sẽ được khởi động lại tự động bởi trình quản lý quy trình được cấu hình trên máy chủ của bạn.

Số giây tối đa mà các tác vụ có thể chạy có thể được chỉ định bằng cách sử dụng công tắc `--timeout` trên dòng lệnh Artisan:

```
php artisan queue:work --timeout=30
```

Nếu tác vụ vượt quá số lần thử tối đa do liên tục hết thời gian, nó sẽ được đánh dấu là không thành công.

Bạn cũng có thể xác định số giây tối đa mà một tác vụ được phép chạy trên chính lớp công việc đó. Nếu thời gian chờ được chỉ định trên tác vụ, nó sẽ được ưu tiên hơn bất kỳ thời gian chờ nào được chỉ định trên dòng lệnh:

```php
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

Đôi khi, các quy trình chặn IO như socket hoặc kết nối HTTP gửi đi có thể không quan trọng thời gian chờ đã chỉ định của bạn. Do đó, khi sử dụng các tính năng này, bạn cũng

nên cố gắng chỉ định thời gian chờ bằng cách sử dụng các API của chúng. Ví dụ: khi sử dụng Guzzle, bạn phải luôn chỉ định kết nối và yêu cầu giá trị thời gian chờ.

Không đúng thời gian chờ

Nếu bạn muốn chỉ ra rằng một tác vụ sẽ được đánh dấu là không thành công khi hết thời gian, bạn có thể xác định thuộc tính `$failOnTimeout` trên class tác vụ:

```
/**
 * Indicate if the job should be marked as failed on timeout.
 *
 * @var bool
 */
public $failOnTimeout = true;
```

Error Handling

Nếu một ngoại lệ được đưa ra trong khi tác vụ đang được xử lý, tác vụ sẽ tự động được đưa trở lại hàng đợi để nó có thể được thử lại. Tác vụ sẽ tiếp tục được phát hành cho đến khi nó đã được thử số lần tối đa mà ứng dụng của bạn cho phép. Số lần thử tối đa được xác định bởi công tắc `--tries` được sử dụng trên lệnh Artisan `queue:work`. Ngoài ra, số lần thử tối đa có thể được xác định trên chính class tác vụ. Bạn có thể tìm thấy thêm thông tin về cách chạy worker hàng chờ bên dưới.

Hoàn thành công việc theo cách thủ công

Đôi khi bạn có thể muốn đưa một tác vụ trở lại hàng chờ theo cách thủ công để có thể thử lại tác vụ sau đó. Bạn có thể thực hiện điều này bằng cách gọi phương thức `release`:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
```

```
{
  // ...

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

Theo mặc định, phương thức `release` sẽ giải phóng tác vụ trở lại hàng chờ để xử lý ngay lập tức. Tuy nhiên, bằng cách truyền một số nguyên cho phương thức `release`, bạn có thể hướng dẫn hàng chờ không cung cấp tác vụ để xử lý cho đến khi trôi qua một số giây nhất định:

```
$this->release(10);
```

Làm thất bại một cách thủ công một tác vụ

Đôi khi, bạn có thể cần phải tự đánh dấu tác vụ là "không thành công". Để làm như vậy, bạn có thể gọi phương thức `fail`:

```
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
  // ...

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

Nếu bạn muốn đánh dấu tác vụ của mình là thất bại vì một ngoại lệ mà bạn đã mắc phải, bạn có thể truyền ngoại lệ cho phương thức `fail`:

```
$this->fail($exception);
```

> Để biết thêm thông tin về các tác vụ thất bại, hãy xem tài liệu về cách xử lý các tác vụ thất bại.

Tác vụ theo batching

Tính năng patching tác vụ của Laravel cho phép bạn dễ dàng thực hiện một loạt tác vụ và sau đó thực hiện một số hành động khi patch tác vụ đã hoàn thành việc thực thi. Trước khi bắt đầu, bạn nên tạo một migration cơ sở dữ liệu để xây dựng một bảng chứa thông tin meta về các patching tác vụ của bạn, chẳng hạn như tỷ lệ phần trăm hoàn thành của chúng. Migration này có thể được tạo bằng cách sử dụng lệnh Artisan `queue:batches-table`:

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

```
php artisan migrate
```

Khai báo tác vụ batchable

Để xác định một tác vụ có thể thực hiện được, bạn nên tạo một tác vụ có thể xếp hàng như bình thường; tuy nhiên, bạn nên thêm trait `Illuminate\Bus\Batchable` vào class tác vụ. Trait này cung cấp quyền truy cập vào một phương thức `batch` có thể được sử dụng để truy xuất batch hiện tại mà tác vụ đang thực thi bên trong:

```php
<?php

namespace App\Jobs;

use Illuminate\Bus\Batchable;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

class ImportCsv implements ShouldQueue
{
  use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;
```

```
    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        if ($this->batch()->cancelled()) {
            // Determine if the batch has been cancelled...

            return;
        }

        // Import a portion of the CSV file...
    }
}
```

Gửi hàng loạt

Để gửi một loạt tác vụ, bạn nên sử dụng phương thức `batch` của facade `Bus`. Tất nhiên, batching chủ yếu hữu ích khi được kết hợp với các lệnh callback hoàn thành. Vì vậy, bạn có thể sử dụng các phương thức `then`, `catch` và `finally` để khai báo callback hoàn thành batch. Mỗi callback này sẽ nhận được một đối tượng `Illuminate\Bus\Batch` khi chúng được gọi. Trong ví dụ này, chúng ta sẽ tưởng tượng rằng chúng ta đang xếp hàng một loạt tác vụ mà mỗi lần xử lý một số hàng nhất định từ tập tin CSV:

```
use App\Jobs\ImportCsv;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;
use Throwable;

$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
```

```
])->then(function (Batch $batch) {
  // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
  // First batch job failure detected...
})->finally(function (Batch $batch) {
  // The batch has finished executing...
})->dispatch();

return $batch->id;
```

ID của batch, có thể được truy cập thông qua thuộc tính `$batch->id`, có thể được sử dụng để truy vấn lệnh `bus` của Laravel để biết thông tin về batch sau khi nó đã được gửi đi.

> **Chú ý:** Vì các callback được batch sẽ được mã hóa và thực thi sau đó bởi hàng chờ của Laravel, bạn không nên sử dụng biến `$this` trong các callback.

Đặt tên các batch

Một số công cụ như Laravel Horizon và Laravel Telescope có thể cung cấp thông tin debug thân thiện với người dùng hơn cho các batch nếu các batch này được đặt tên. Để gán một tên tùy ý cho một batch, bạn có thể gọi phương thức `name` trong khi khai báo batch:

```
$batch = Bus::batch([
  // ...
])->then(function (Batch $batch) {
  // All jobs completed successfully...
})->name('Import CSV')->dispatch();
```

Kết nối & xếp hàng chờ batch

Nếu bạn muốn chỉ định kết nối và hàng đợi sẽ được sử dụng cho các tác vụ theo đợt, bạn có thể sử dụng các phương thức `onConnection` và `onQueue`. Tất cả các tác vụ theo đợt phải thực thi trong cùng một kết nối và hàng chờ:

```
$batch = Bus::batch([
  // ...
```

```
])->then(function (Batch $batch) {
  // All jobs completed successfully...
})->onConnection('redis')->onQueue('imports')->dispatch();
```

Chuỗi trong batch

Bạn có thể khai báo một tập hợp các tác vụ được xâu chuỗi trong một batch bằng cách đặt các tác vụ được xâu chuỗi trong một mảng. Ví dụ: chúng tôi có thể thực hiện song song hai chuỗi tác vụ và thực hiện lệnh gọi lại khi cả hai chuỗi tác vụ đã xử lý xong:

```
use App\Jobs\ReleasePodcast;
use App\Jobs\SendPodcastReleaseNotification;
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

Bus::batch([
  [
    new ReleasePodcast(1),
    new SendPodcastReleaseNotification(1),
  ],
  [
    new ReleasePodcast(2),
    new SendPodcastReleaseNotification(2),
  ],
])->then(function (Batch $batch) {
  // ...
})->dispatch();
```

Thêm tác vụ vào batch

Đôi khi, việc thêm các tác vụ bổ sung vào một batch từ bên trong một tác vụ theo batch có thể hữu ích. Mẫu này có thể hữu ích khi bạn cần gộp hàng nghìn tác vụ có thể mất quá nhiều thời gian để gửi đi trong một web request. Vì vậy, thay vào đó, bạn có thể muốn để gửi một loạt tác vụ "loader" với nhiều tác vụ hơn:

```
$batch = Bus::batch([
```

```
    new LoadImportBatch,
    new LoadImportBatch,
    new LoadImportBatch,
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->name('Import Contacts')->dispatch();
```

Trong ví dụ này, chúng ta sẽ sử dụng tác vụ `LoadImportBatch` để chạy batch với các tác vụ bổ sung. Để thực hiện điều này, chúng ta có thể sử dụng phương thức `add` trên đối tượng batch có thể được truy cập thông qua phương thức `batch` của tác vụ:

```
use App\Jobs\ImportContacts;
use Illuminate\Support\Collection;

/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
        return new ImportContacts;
    }));
}
```

Chú ý: Bạn chỉ có thể thêm tác vụ vào một batch từ bên trong một tác vụ thuộc cùng một batch.

```
// The UUID of the batch...
$batch->id;
```

```php
// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;

// The number of jobs that have failed...
$batch->failedJobs;

// The number of jobs that have been processed thus far...
$batch->processedJobs();

// The completion percentage of the batch (0-100)...
$batch->progress();

// Indicates if the batch has finished executing...
$batch->finished();

// Cancel the execution of the batch...
$batch->cancel();

// Indicates if the batch has been cancelled...
$batch->cancelled();
```

Trả lại batch từ các route

Tất cả các đối tượng `Illuminate\Bus\Batch` đều có thể mã hóa JSON, nghĩa là bạn có thể trả lại chúng trực tiếp từ một trong các route của ứng dụng của mình để truy xuất tải trọng JSON chứa thông tin về batch, bao gồm cả tiến độ hoàn thành của batch. Điều này giúp thuận tiện trong việc hiển thị thông tin về việc hoàn thành batch tiến trình trong giao diện người dùng của ứng dụng của bạn.

Để truy xuất một batch theo ID của nó, bạn có thể sử dụng phương thức `findBatch` của facade `Bus`:

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

Route::get('/batch/{batchId}', function (string $batchId) {
  return Bus::findBatch($batchId);
});
```

Hủy batch

Đôi khi bạn có thể cần phải hủy thực thi một batch nào đó. Điều này có thể được thực hiện bằng cách gọi phương thức cancel trên đối tượng `Illuminate\Bus\Batch`:

```php
/**
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
  if ($this->user->exceedsImportLimit()) {
    return $this->batch()->cancel();
  }

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

Như bạn có thể đã nhận thấy trong các ví dụ trước, các tác vụ theo batch thường nên kiểm tra xem liệu batch đã bị hủy ở đầu phương thức `handle` của chúng hay chưa:

```php
/**
 * Execute the job.
 *
 * @return void
```

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

  // Continue processing...
}
```

Lỗi trong batch

Khi một tác vụ theo batch không thành công, lệnh callback `catch` (nếu được chỉ định) sẽ được gọi. Lệnh callback này chỉ được gọi cho tác vụ đầu tiên không thành công trong batch.

Cho phép thất bại

Khi một tác vụ trong một batch không thành công, Laravel sẽ tự động đánh dấu batch đó là "cancel". Nếu muốn, bạn có thể vô hiệu hóa hành vi này để lỗi tác vụ không tự động đánh dấu batch là đã bị hủy. Điều này có thể được thực hiện bằng cách gọi `allowFailures` trong khi gửi batch:

```
$batch = Bus::batch([
  // ...
])->then(function (Batch $batch) {
  // All jobs completed successfully...
})->allowFailures()->dispatch();
```

Thử lại Hàng loạt Tác vụ Không thành công

Để thuận tiện, Laravel cung cấp một lệnh Artisan `queue:retry-batch` cho phép bạn dễ dàng thử lại tất cả các tác vụ bị lỗi cho một batch nhất định. Lệnh `queue:retry-batch` chấp nhận UUID của batch có các tác vụ bị lỗi sẽ được thử lại:

```
php artisan queue:retry-batch 32dbc76c-4f82-4749-b610-a639fe0099b5
```

Giải phóng batch

Nếu không giải phóng batch, bảng `job_batches` có thể tích lũy các record rất nhanh. Để giảm thiểu điều này, bạn nên lên lịch chạy lệnh Artisan `queue:trimne-batches` hàng ngày:

```
$schedule->command('queue:prune-batches')->daily();
```

Theo mặc định, tất cả các batch đã hoàn thành trong 24 giờ sẽ bị giải phóng. Bạn có thể sử dụng tùy chọn giờ khi gọi lệnh để xác định thời gian lưu giữ dữ liệu batch. Ví dụ: lệnh sau sẽ xóa tất cả các batch đã hoàn thành trong 48 giờ trước đó:

```
$schedule->command('queue:prune-batches --hours=48')->daily();
```

Đôi khi, bảng `job_batches` của bạn có thể tích lũy các record batch đối với các batch không bao giờ hoàn thành, chẳng hạn như các batch mà một tác vụ không thành công và tác vụ đó không bao giờ được thử lại thành công. Bạn có thể hướng dẫn lệnh `queue:trimne-batches` để loại bỏ các record batch chưa hoàn thành này bằng cách sử dụng tùy chọn chưa hoàn thành:

```
$schedule->command('queue:prune-batches --hours=48 --unfinished=72')->daily();
```

Xếp hàng chờ cho hàm nặc danh

Thay vì điều động một class tác vụ vào hàng đợi, bạn cũng có thể gửi một kết thúc. Điều này rất tốt cho các tác vụ nhanh chóng, đơn giản cần được thực hiện bên ngoài chu kỳ yêu cầu hiện tại. Khi gửi các hàm nặc danh đến hàng chờ, nội dung mã của hàm nặc danh được ký bằng mật mã để không thể sửa đổi khi chuyển tiếp:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
  $podcast->publish();
});
```

Khi sử dụng phương thức `catch`, bạn có thể cung cấp một hàm nặc danh mà sẽ được thực

thi nếu các hàm được lên hàng chờ này không hoàn thành thành công sau khi hoàn thành tất cả các lần thử lại đã được cấu hình trong cấu hình hàng chờ của bạn:

```
use Throwable;

dispatch(function () use ($podcast) {
  $podcast->publish();
})->catch(function (Throwable $e) {
  // This job has failed...
});
```

Chạy Worker Hàng chờ

Lệnh `queue:work`

Laravel bao gồm một lệnh Artisan sẽ bắt đầu một worker hàng chờ và xử lý các tác vụ mới khi chúng được đẩy vào hàng chờ. Bạn có thể chạy worker bằng lệnh Artisan `queue:work`. Lưu ý rằng khi lệnh `queue:work` đã bắt đầu, nó sẽ tiếp tục chạy cho đến khi nó bị dừng theo cách thủ công hoặc bạn đóng terminal của mình:

```
php artisan queue:work
```

> Để giữ cho tiến trình lệnh queue:work chạy vĩnh viễn trong nền, bạn nên sử dụng bộ giám sát quá trình như Supervisor để đảm bảo rằng worker hàng chờ không ngừng chạy.

Hãy nhớ rằng, các worker hàng chờ là các tiến trình tồn tại lâu dài và lưu trữ trạng thái ứng dụng đã khởi động trong bộ nhớ. Do đó, chúng sẽ không nhận thấy những thay đổi trong cơ sở mã của bạn sau khi chúng đã được khởi động. Vì vậy, trong quá trình triển khai, bạn hãy đảm bảo khởi động lại worker hàng chờ của bạn. Ngoài ra, hãy nhớ rằng bất kỳ trạng thái tĩnh nào được tạo hoặc sửa đổi bởi ứng dụng của bạn sẽ không được tự động được đặt lại giữa các tác vụ.

Ngoài ra, bạn có thể chạy lệnh `queue:listen`. Khi sử dụng lệnh `queue:listen`, bạn không phải khởi động lại worker theo cách thủ công khi bạn muốn tải lại mã đã cập nhật của mình hoặc đặt lại trạng thái ứng dụng; tuy nhiên, lệnh này kém hiệu quả một cách đáng kể so với lệnh `queue:work`:

```
php artisan queue:listen
```

Chạy nhiều worker hàng chờ

Để chỉ định nhiều worker vào một hàng chờ và xử lý tác vụ một cách đồng bộ, bạn chỉ cần bắt đầu nhiều tiến trình `queue:work`. Điều này có thể được thực hiện cục bộ thông qua nhiều tab trong terminal của bạn hoặc trong quá trình sản xuất bằng cách sử dụng cài đặt cấu hình của trình quản lý quy trình của bạn. Khi sử dụng Supervisor, bạn có thể sử dụng giá trị cấu hình `numprocs`.

Chỉ định Kết nối & Hàng chờ

Bạn cũng có thể chỉ định kết nối hàng chờ mà worker sẽ sử dụng. Tên kết nối được truyền cho lệnh `work` phải tương ứng với một trong các kết nối được khai báo trong tập tin cấu hình *config/queue.php* của bạn:

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

Theo mặc định, lệnh `queue:work` sẽ chỉ xử lý các tác vụ cho hàng chờ mặc định trên một kết nối nào đó. Tuy nhiên, bạn có thể tùy chỉnh nâng cao hơn cho worker hàng chờ bằng việc cho nó chỉ xử lý các hàng chờ cụ thể thay cho hàng chờ mặc định trên một kết nối nào đó. Ví dụ: nếu tất cả các email của bạn phải xử lý hàng chờ email trên kết nối hàng đợi redis, thì bạn có thể ra lệnh sau để bắt đầu một worker:

```
php artisan queue:work redis --queue=emails
```

Xử lý một số tác vụ cụ thể

Tùy chọn `--once` có thể được sử dụng để hướng dẫn worker chỉ xử lý một tác vụ đơn nhất từ hàng chờ:

```
php artisan queue:work --once
```

Tùy chọn `--max-jobs` có thể được sử dụng để cho worker biết số lượng tác vụ được xử lý và thoát sau đó. Tùy chọn này có thể hữu ích khi được kết hợp với Supervisor để worker của bạn tự động khởi động lại sau khi xử lý một số công việc nhất định, giải phóng bất kỳ bộ nhớ nào mà họ có thể đã tích lũy:

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

Xử lý tất cả công việc đã xếp hàng và thoát sau đó

Tùy chọn `--stop-when-empty` có thể được sử dụng để cho worker xử lý tất cả các tác vụ và thoát sau đó. Tùy chọn này có thể hữu ích khi xử lý hàng chờ Laravel trong container Docker nếu bạn muốn tắt container sau khi hàng chờ trống:

```
php artisan queue:work --stop-when-empty
```

Xử lý tác vụ trong một số giây nhất định

Tùy chọn `--max-time` có thể được sử dụng để hướng dẫn worker xử lý tác vụ trong một số giây nhất định và thoát sau đó. Tùy chọn này có thể hữu ích khi được kết hợp với Supervisor để worker của bạn tự động khởi động lại sau khi xử lý tác vụ trong một khoảng thời gian nhất định, và giải phóng bất kỳ bộ nhớ nào mà chúng đã tích lũy:

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

Thời lượng nghỉ của worker

Khi các tác vụ đã sẵn sàng trên hàng chờ, worker sẽ tiếp tục xử lý các tác vụ mà không có sự chậm trễ nào giữa chúng. Tuy nhiên, tùy chọn sleep sẽ cho worker sẽ "nghỉ" bao nhiêu giây nếu không có tác vụ mới. Trong khi nghỉ, worker sẽ không xử lý bất kỳ tác vụ mới nào - các tác vụ sẽ được xử lý sau khi worker thức tỉnh trở lại.

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

Những chú ý về tài nguyên

Worker hàng chờ Daemon sẽ không "reboot" lại framework trước khi xử lý mỗi tác vụ. Do đó, bạn nên giải phóng bất kỳ tài nguyên nào sau khi tác vụ hoàn thành. Ví dụ: nếu bạn đang thực hiện thao tác hình ảnh với thư viện GD, bạn nên giải phóng bộ nhớ bằng `imagedestroy` khi bạn xử lý xong hình ảnh.

Ưu tiên trong hàng chờ

Đôi khi bạn có thể muốn ưu tiên cách mà hàng chờ của mình được xử lý. Ví dụ: trong tập tin cấu hình *config/queue.php* của bạn, bạn có thể đặt cấu hình `queue` của mình thành `low`. Tuy nhiên, bạn cũng có thể muốn đẩy một tác vụ lên hàng chờ ưu tiên là `high` như sau:

```
dispatch((new Job)->onQueue('high'));
```

Để bắt đầu một worker mà xác minh rằng tất cả các tác vụ trên hàng chờ `high` đều được xử lý trước khi tiếp tục bất kỳ tác vụ nào trên hàng chờ `low`, hãy truyền danh sách tên hàng chờ được phân tách bằng dấu phẩy vào lệnh `work`:

```
php artisan queue:work --queue=high,low
```

Worker hàng chờ & Triển khai hàng chờ

Vì worker hàng chờ là các quy trình tồn tại lâu dài, nên chúng sẽ không nhận thấy các thay đổi đối với mã của bạn nếu không được khởi động lại. Vì vậy, cách đơn giản nhất để triển khai một ứng dụng sử dụng worker hàng chờ là khởi động lại worker trong quá trình deploy của bạn. Bạn có thể khởi động lại tất cả các worker một cách nhanh gọn bằng cách đưa ra lệnh `queue:restart`:

```
php artisan queue:restart
```

Lệnh này sẽ hướng dẫn tất cả worker hàng chờ thoát một cách gọn gàng sau khi họ xử lý xong tác vụ hiện tại để không có tác vụ hiện có nào bị mất. Vì các worker hàng chờ sẽ thoát ra khi lệnh `queue:restart` được thực thi, bạn nên chạy một trình quản lý tiến trình như Supervisor để tự động khởi động lại các worker hàng chờ.

> Hàng chờ sử dụng bộ đệm để lưu các tín hiệu khởi động lại, vì vậy bạn nên xác minh rằng driver bộ đệm được cấu hình đúng cho ứng dụng của bạn trước khi sử dụng tính năng này.

Gia hạn tác vụ & thời hạn tác vụ

Gia hạn tác vụ

Trong tập tin cấu hình *config/queue.php* của bạn, mỗi kết nối hàng đợi sẽ khai báo một tùy chọn `retry_after`. Tùy chọn này chỉ định kết nối hàng chờ sẽ đợi bao nhiêu giây trước khi thử lại một tác vụ đang được xử lý. Ví dụ: nếu giá trị của `retry_after` được đặt thành

`90`, tác vụ sẽ được giải phóng trở lại hàng chờ nếu nó đã được xử lý trong `90` giây mà không bị giải phóng hoặc xóa. Thông thường, bạn nên đặt giá trị `retry_after` thành số giây tối đa mà tác vụ của bạn phải thực hiện một cách hợp lý để hoàn tất quá trình xử lý.

> **Chú ý:** Kết nối hàng đợi duy nhất không chứa giá trị `retry_after` là Amazon SQS. SQS sẽ thử lại tác vụ dựa trên Default Visibility Timeout được quản lý trong bảng điều khiển AWS.

Thời hạn của worker

Lệnh Artisan `queue:work` có đưa ra tùy chọn `--timeout`. Theo mặc định, giá trị `--timeout` là 60 giây. Nếu một tác vụ đang xử lý lâu hơn số giây được chỉ định bởi giá trị timeout, worker đang xử lý tác vụ sẽ thoát ra với một lỗi. Thông thường, worker sẽ được khởi động lại tự động bởi trình quản lý quy trình được cấu hình trên máy chủ của bạn:

```
php artisan queue:work --timeout=60
```

Tùy chọn cấu hình `retry_after` và tùy chọn cli `--timeout` là hai tùy chọn khác nhau, nhưng hoạt động cùng nhau để đảm bảo các tác vụ không bị mất và các tác vụ chỉ được xử lý thành công một lần.

> **Chú ý:** Giá trị `--timeout` luôn phải ngắn hơn giá trị cấu hình `retry_after` của bạn ít nhất vài giây. Điều này sẽ đảm bảo rằng một worker đang xử lý tác vụ bị đóng băng luôn được kết thúc trước khi tác vụ được thử lại. Nếu tùy chọn `--timeout` của bạn dài hơn giá trị cấu hình `retry_after` của bạn, thì tác vụ của bạn có thể được xử lý hai lần.

Cấu hình Supervisor

Trong quá trình sản xuất, bạn cần một cách để giữ cho tiến trình `queue:work` của mình luôn chạy. Tiến trình `queue:work` có thể ngừng chạy vì nhiều lý do, chẳng hạn như timeout của worker đã vượt quá hoặc việc thực thi lệnh `queue:restart`.

Vì lý do này, bạn cần cấu hình chương trình giám sát quy trình có thể phát hiện khi tiến trình xử lý lệnh `queue:work` thoát và tự động khởi động lại chúng. Ngoài ra, chương trình theo dõi quy trình có thể cho phép bạn chỉ định bao nhiêu tiến trình `queue:work` mà bạn muốn chạy đồng thời. Supervisor là một chương trình giám sát quá trình thường được sử dụng trong môi trường Linux và chúng ta sẽ thảo luận về cách cấu hình nó trong tài liệu sau.

Cài đặt Supervisor

Supervisor là một trình giám sát quá trình cho hệ điều hành Linux và sẽ tự động khởi động lại tiến trình queue:work của bạn nếu chúng bị lỗi. Để cài đặt Supervisor trên Ubuntu, bạn có thể sử dụng lệnh sau:

```
sudo apt-get install supervisor
```

> Nếu việc tự cấu hình và quản lý Supervisor nghe có vẻ quá sức, hãy cân nhắc sử dụng Laravel Forge, ứng dụng này sẽ tự động cài đặt và cấu hình Supervisor cho các dự án Laravel của bạn.

Cấu hình Supervisor

Các tập tin cấu hình cho Supervisor thường được lưu trữ trong thư mục `/etc/supervisor/conf.d`. Trong thư mục này, bạn có thể tạo bất kỳ số lượng tập tin cấu hình nào hướng dẫn supervisor cách giám sát các quy trình của bạn. Ví dụ: hãy tạo một tập tin *laravel-worker.conf* mà sẽ khởi động và giám sát các quy trình `queue:work`:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3 --max-time=3600
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
stopwaitsecs=3600
```

Trong ví dụ này, lệnh `numprocs` sẽ chỉ dẫn Supervisor chạy tám tiến trình `queue:work` và giám sát tất cả chúng, cũng như tự động khởi động lại chúng nếu chúng không thành công. Bạn nên thay đổi chỉ thị `command` của cấu hình để phản ánh kết nối hàng chờ mong muốn của bạn và các tùy chọn của worker.

> **Chú ý:** Bạn nên đảm bảo rằng giá trị của `stopwaitsec` lớn hơn số giây được sử

dụng bởi tác vụ đang chạy lâu nhất của bạn. Nếu không, Supervisor có thể giết tác vụ trước khi nó được xử lý xong.

Bắt đầu starting

Khi tập tin cấu hình đã được tạo, bạn có thể cập nhật cấu hình Supervisor và bắt đầu các quy trình bằng các lệnh sau:

```
sudo supervisorctl reread
```

```
sudo supervisorctl update
```

```
sudo supervisorctl start laravel-worker:*
```

Để biết thêm thông tin về Supervisor, hãy tham khảo tài liệu Supervisor.

Xử lý công việc thất bại

Đôi khi tác vụ trên hàng chờ của bạn sẽ thất bại. Đừng lo lắng, mọi thứ không phải lúc nào cũng diễn ra như kế hoạch! Laravel có đưa ra một cách thuận tiện để chỉ định số lần tối đa mà một công việc nên được thực hiện. Sau khi một tác vụ không đồng bộ vượt quá số lần thử này, nó sẽ được chèn vào bảng cơ sở dữ liệu `failed_jobs`. Các tác vụ được gửi đồng bộ bị lỗi không được lưu trữ trong bảng này và các exception của chúng sẽ được ứng dụng xử lý ngay lập tức.

Một migration để tạo bảng `failed_jobs` thường được đặt sẵn trong các ứng dụng Laravel mới. Tuy nhiên, nếu ứng dụng của bạn không chứa migration cho bảng này, bạn có thể sử dụng lệnh `queue:failed-table` để tạo migration:

```
php artisan queue:failed-table
```

```
php artisan migrate
```

Khi chạy quy trình worker hàng chờ, bạn có thể chỉ định số lần tối đa một tác vụ sẽ được thực hiện bằng cách sử dụng công tắc `--tries` trên lệnh `queue:work`. Nếu bạn không chỉ định một giá trị cho tùy chọn `--tries`, các tác vụ sẽ chỉ được thực hiện một lần hoặc nhiều lần như được chỉ định bởi thuộc tính `$try` của class tác vụ:

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

Sử dụng tùy chọn `--backoff`, bạn có thể chỉ định Laravel sẽ đợi bao nhiêu giây trước khi thử lại một tác vụ gặp phải exception. Theo mặc định, một tác vụ ngay lập tức được đưa trở lại hàng chờ để nó có thể được thử lại:

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

Nếu bạn muốn cấu hình Laravel để nó đợi bao nhiêu giây trước khi thử lại một tác vụ gặp phải exception trên cơ sở từng tác vụ, bạn có thể thực hiện việc này bằng cách xác định một thuộc tính `backoff` trên class tác vụ của mình:

```
/**
 * The number of seconds to wait before retrying the job.
 *
 * @var int
 */
public $backoff = 3;
```

Nếu bạn yêu cầu logic phức tạp hơn để xác định thời gian `backoff` của tác vụ, bạn có thể khai báo một phương thức `backoff` trên class tác vụ của mình:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return int
*/
public function backoff()
{
  return 3;
}
```

Bạn có thể dễ dàng cấu hình các backoff "exponential"(tiềm năng) bằng cách trả về một mảng các giá trị backoff từ phương thức `backoff`. Trong ví dụ này, độ trễ khi thử lại sẽ là 1 giây cho lần thử lại đầu tiên, 5 giây cho lần thử lại thứ hai và 10 giây cho lần thử lại thứ ba:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
```

```
     * @return array
     */
    public function backoff()
    {
        return [1, 5, 10];
    }
```

Dọn dẹp sau khi tác vụ thất bại

Khi một tác vụ cụ thể không thành công, bạn có thể sẽ muốn gửi cảnh báo cho người dùng của mình hoặc hoàn nguyên bất kỳ hành động nào đã được hoàn thành một phần bởi tác vụ. Để thực hiện điều này, bạn có thể khai báo phương thức `failed` trên tác vụ của mình. Đối tượng `Throwable` sẽ khiến tác vụ không thành công được truyền vào phương thức `failed`:

```
<?php

namespace App\Jobs;

use App\Models\Podcast;
use App\Services\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;
use Throwable;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The podcast instance.
     *
     * @var \App\Podcast
     */
    public $podcast;
```

```
    /**
     * Create a new job instance.
     *
     * @param  \App\Models\Podcast  $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param  \App\Services\AudioProcessor  $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }

    /**
     * Handle a job failure.
     *
     * @param  \Throwable  $exception
     * @return void
     */
    public function failed(Throwable $exception)
    {
        // Send user notification of failure, etc...
    }
}
```

Chú ý: Một đối tượng mới của tác vụ được khởi tạo trước khi gọi phương thức `failed`; do đó, bất kỳ sửa đổi thuộc tính nào của class có thể đã xảy ra bên trong

phương thức `handle` sẽ bị mất.

Thử lại tác vụ failed

Để xem tất cả các tác vụ failed đã được chèn vào bảng cơ sở dữ liệu `failed_jobs` của bạn, bạn có thể sử dụng lệnh Artisan `queue:failed`:

```
php artisan queue:failed
```

Lệnh queue:failed sẽ liệt kê ID tác vụ, kết nối, hàng chờ, thời gian thất bại và các thông tin khác về tác vụ. ID tác vụ có thể được sử dụng để thử lại tác vụ không thành công. Ví dụ: để thử lại một tác vụ không thành công có ID là `ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece`, hãy chạy lệnh sau:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece
```

Nếu cần, bạn có thể truyền nhiều ID vào lệnh:

```
php artisan queue:retry ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece 91401d2c-0784-4f43-824c-34f94a33c24d
```

Bạn cũng có thể thử lại tất cả các tác vụ không thành công cho một hàng chờ cụ thể:

```
php artisan queue:retry --queue=name
```

Để thử lại tất cả các tác vụ không thành công của bạn, hãy thực hiện lệnh `queue:retry` và truyền tất cả dưới dạng ID:

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

Nếu bạn muốn xóa một tác vụ không thành công, thì bạn có thể sử dụng lệnh `queue:forget`:

```
php artisan queue:forget 91401d2c-0784-4f43-824c-34f94a33c24d
```

> Khi sử dụng Horizon, bạn nên sử dụng lệnh horizon:forget để xóa tác vụ không thành công thay vì lệnh `queue:forget`.

Để xóa tất cả các tác vụ không thành công của bạn khỏi bảng `failed_jobs`, bạn có thể sử

dụng lệnh `queue:flush`:

```
php artisan queue:flush
```

Bỏ qua các mô hình bị thiếu

Khi đưa một mô hình Eloquent vào một tác vụ, mô hình sẽ tự động được mã hóa trước khi được đặt vào hàng chờ và được truy xuất lại từ cơ sở dữ liệu khi tác vụ được xử lý. Tuy nhiên, nếu mô hình đã bị xóa trong khi tác vụ đang chờ xử lý bởi worker, thì tác vụ của bạn có thể không thành công với `ModelNotFoundException`.

Để thuận tiện, bạn có thể chọn tự động xóa tác vụ với các mô hình bị thiếu bằng cách đặt thuộc tính `deleteWhenMissingModels` của tác vụ thành `true`. Khi thuộc tính này được đặt thành `true`, thì Laravel sẽ lặng lẽ loại bỏ tác vụ mà không đưa ra exception:

```
/**
 * Delete the job if its models no longer exist.
 *
 * @var bool
 */
public $deleteWhenMissingModels = true;
```

Lượt bớt các tác vụ failed

Bạn có thể xóa tất cả các record trong bảng `failed_jobs` trong ứng dụng của mình bằng cách gọi lệnh Artisan `queue:prune-failed`:

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

Nếu bạn cung cấp tùy chọn `--hours` cho lệnh, chỉ những record tác vụ bị lỗi đã được chèn trong N số giờ đã qua mới được giữ lại. Ví dụ: lệnh sau sẽ xóa tất cả các record tác vụ bị lỗi đã được chèn trong hơn 48 giờ trước:

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

Lưu trữ Tác vụ Không thành công trong DynamoDB

Laravel cũng hỗ trợ lưu trữ hồ sơ tác vụ bị lỗi của bạn trong DynamoDB thay vì bảng cơ sở dữ liệu. Tuy nhiên, bạn phải tạo một bảng DynamoDB để lưu trữ tất cả các record tác vụ bị lỗi. Thông thường, bảng này phải được đặt tên là `failed_jobs`, nhưng bạn nên đặt tên bảng dựa trên giá trị của giá trị cấu hình `queue.failed.table` trong tập tin cấu hình hàng chờ của ứng dụng của bạn.

Bảng `failed_jobs` phải có khóa phân vùng chính kiểu chuỗi có tên là `application` và một khóa xếp thứ tự chính kiểu chuỗi có tên là `uuid`. Phần `application` của khóa sẽ chứa tên ứng dụng của bạn như được xác định bởi giá trị cấu hình `name` trong tập tin cấu hình `app` của ứng dụng của bạn. Do tên ứng dụng là một phần của khóa trong bảng DynamoDB, bạn có thể sử dụng cùng một bảng để lưu trữ các tác vụ không thành công cho nhiều ứng dụng Laravel.

Ngoài ra, hãy đảm bảo rằng bạn cài đặt AWS SDK để ứng dụng Laravel của bạn có thể giao tiếp với Amazon DynamoDB:

```
composer require aws/aws-sdk-php
```

Tiếp theo, đặt giá trị của tùy chọn cấu hình `queue.failed.driver` thành `dynamicodb`. Ngoài ra, bạn nên khai báo các tùy chọn cấu hình key, secret và region trong mảng cấu hình tác vụ không thành công. Các tùy chọn này sẽ được sử dụng để xác thực với AWS. Khi sử dụng driver `dynamicodb`, tùy chọn cấu hình `queue.failed.database` là không cần thiết:

```
'failed' => [
  'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
  'key' => env('AWS_ACCESS_KEY_ID'),
  'secret' => env('AWS_SECRET_ACCESS_KEY'),
  'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
  'table' => 'failed_jobs',
],
```

Tắt lưu trữ tác vụ không thành công

Bạn có thể hướng dẫn Laravel loại bỏ các tác vụ bị lỗi mà không cần lưu trữ chúng bằng cách đặt giá trị của tùy chọn cấu hình `queue.failed.driver` thành `null`. Thông thường, điều này có thể được thực hiện thông qua biến môi trường `QUEUE_FAILED_DRIVER`:

```
QUEUE_FAILED_DRIVER=null
```

Sự kiện của tác vụ không thành công

Nếu bạn muốn đăng ký một chương trình theo dõi event không thành công cho các tác vụ, thì bạn có thể sử dụng phương thức `failing` của facade `Queue`. Ví dụ: chúng ta có thể đính kèm một hàm callback cho event này từ phương thức `boot` của `AppServiceProvider` mà được bao gồm trong Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobFailed;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Register any application services.
   *
   * @return void
   */
  public function register()
  {
    //
  }

  /**
   * Bootstrap any application services.
   *
   * @return void
   */
  public function boot()
  {
```

```
    Queue::failing(function (JobFailed $event) {
      // $event->connectionName
      // $event->job
      // $event->exception
    });
  }
}
```

Xóa tác vụ khỏi hàng chờ

> Khi sử dụng Horizon, bạn nên sử dụng lệnh Horizon: clear để xóa các công việc khỏi hàng đợi thay vì lệnh queue: clear.

Nếu bạn muốn xóa tất cả các tác vụ khỏi hàng chờ mặc định của kết nối mặc định, bạn có thể làm như vậy bằng cách sử dụng lệnh Artisan `queue:clear`:

```
php artisan queue:clear
```

Bạn cũng có thể cung cấp đối số kết nối và tùy chọn hàng chờ để xóa tác vụ khỏi một kết nối và hàng đợi cụ thể:

```
php artisan queue:clear redis --queue=emails
```

> **Chú ý:** Xóa tác vụ khỏi hàng chờ chỉ khả dụng cho driver hàng đợi SQS, Redis và cơ sở dữ liệu. Ngoài ra, quá trình xóa tin nhắn SQS mất tới 60 giây, do đó, các tác vụ được gửi đến hàng đợi SQS lên đến 60 giây sau khi bạn xóa hàng chờ cũng có thể bị xóa.

Theo dõi hàng chờ của bạn

Nếu hàng chờ của bạn nhận được một lượng tác vụ đột ngột, nó có thể trở nên quá tải, dẫn đến thời gian chờ đợi lâu để hoàn thành tác vụ. Nếu bạn muốn, Laravel có thể cảnh báo bạn khi số lượng tác vụ trong hàng chờ của bạn đã vượt quá ngưỡng được quy định.

Để bắt đầu, bạn nên lên lịch để lệnh `queue:monitor` chạy mỗi phút. Lệnh chấp nhận tên của hàng chờ mà bạn muốn theo dõi cũng như ngưỡng số lượng tác vụ mong muốn của bạn:

```
php artisan queue:monitor redis:default,redis:deployments --max=100
```

Chỉ lên lịch lệnh này thôi là không đủ để kích hoạt thông báo cảnh báo bạn về tình trạng quá tải của hàng chờ. Khi lệnh gặp hàng chờ có số lượng tác vụ vượt quá ngưỡng của bạn, thì event `Illuminate\Queue\Events\QueueBusy` sẽ được gửi đi. Bạn có thể theo dõi event này trong `EventServiceProvider` của ứng dụng để gửi thông báo cho bạn hoặc nhóm phát triển của bạn:

```
use App\Notifications\QueueHasLongWaitTime;
use Illuminate\Queue\Events\QueueBusy;
use Illuminate\Support\Facades\Event;
use Illuminate\Support\Facades\Notification;

/**
 * Register any other events for your application.
 *
 * @return void
 */
public function boot()
{
  Event::listen(function (QueueBusy $event) {
    Notification::route('mail', 'dev@example.com')
      ->notify(new QueueHasLongWaitTime(
                $event->connection,
                $event->queue,
                $event->size
    ));
  });
}
```

Các event của tác vụ

Sử dụng các phương thức `before` và `after` trên facade Queue, bạn có thể chỉ định các lệnh callback được thực thi trước hoặc sau khi xử lý tác vụ đã xếp hàng. Các callbacks này là cơ hội tuyệt vời để thực hiện ghi log bổ sung hoặc thống kê cho dashboard. Thông thường, bạn nên gọi các phương thức này từ phương thức `boot` của service provider. Ví dụ: chúng tôi có thể sử dụng `AppServiceProvider` được bao gồm trong Laravel:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Register any application services.
   *
   * @return void
   */
  public function register()
  {
    //
  }

  /**
   * Bootstrap any application services.
   *
   * @return void
   */
  public function boot()
  {
    Queue::before(function (JobProcessing $event) {
      // $event->connectionName
      // $event->job
      // $event->job->payload()
    });

    Queue::after(function (JobProcessed $event) {
      // $event->connectionName
      // $event->job
```

```
      // $event->job->payload()
    });
  }
}
```

Sử dụng phương thức `looping` trên facade `Queue`, bạn có thể chỉ định các callback thực thi trước khi worker cố gắng tìm và fetch một tác vụ từ một hàng chờ. Ví dụ: bạn có thể đăng ký một hàm nặc danh để khôi phục bất kỳ giao dịch nào bị bỏ ngỏ bởi một tác vụ không thành công trước đó:

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

Queue::looping(function () {
  while (DB::transactionLevel() > 0) {
    DB::rollBack();
  }
});
```