VERSION

8.x

Search Docs (Press '/')

Queues

Introduction

While building your web application, you may have some tasks, such as parsing and storing an uploaded CSV file, that take too long to perform during a typical web request. Thankfully, Laravel allows you to easily create queued jobs that may be processed in the background. By moving time intensive tasks to a queue, your application can respond to web requests with blazing speed and provide a better user experience to your customers.

Laravel queues provide a unified queueing API across a variety of different queue backends, such as Amazon SQS, Redis, or even a relational database.

Laravel's queue configuration options are stored in your application's `config/queue.php` configuration file. In this file, you will find connection configurations for each of the queue drivers that are included with the framework, including the database, Amazon SQS, Redis, and Beanstalkd drivers, as well as a synchronous driver that will execute jobs immediately (for use during local development). A `null` queue driver is also included which discards queued jobs.

Laravel now offers Horizon, a beautiful dashboard and configuration system for your Redis powered queues. Check out the full Horizon documentation for more information.

Connections Vs. Queues

Before getting started with Laravel queues, it is important to understand the distinction between "connections" and "queues". In your `config/queue.php` configuration file, there is a `connections` configuration array. This option defines the connections to backend queue services such as Amazon SQS, Beanstalk, or Redis. However, any given queue connection may have multiple "queues" which may be thought of as different stacks or piles of queued jobs.

Note that each connection configuration example in the `queue` configuration file contains a `queue` attribute. This is the default queue that jobs will be dispatched to when they are sent to a given connection. In other words, if you dispatch a job without explicitly defining which queue it should be dispatched to, the job will be placed on the queue that is defined in the `queue` attribute of the connection configuration:

```
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');
```

Some applications may not need to ever push jobs onto multiple queues, instead preferring to have one simple queue. However, pushing jobs to multiple queues can be especially useful for applications that wish to prioritize or segment how jobs are processed, since the Laravel queue worker allows you to specify which queues it should process by priority. For example, if you push jobs to a `high` queue, you may run a worker that gives them higher processing priority:

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

Driver Notes & Prerequisites

Database

In order to use the `database` queue driver, you will need a database table to hold the jobs. To generate a migration that creates this table, run the `queue:table` Artisan command. Once the migration has been created, you may migrate your database using the `migrate` command:

```
php artisan queue:table

php artisan migrate
```

Redis

In order to use the `redis` queue driver, you should configure a Redis database connection in your `config/database.php` configuration file.

Redis Cluster

If your Redis queue connection uses a Redis Cluster, your queue names must contain a key hash tag. This is required in order to ensure all of the Redis keys for a given queue are placed into the same hash slot:

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

Blocking

When using the Redis queue, you may use the `block_for` configuration option to specify how long the driver should wait for a job to become available before iterating through the worker loop and re-polling the Redis database.

Adjusting this value based on your queue load can be more efficient than continually polling the Redis database for new jobs. For instance, you may set the value to `5` to indicate that the driver should block for five seconds while waiting for a job to become available:

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

```
    'block_for' => 5,
],
```

Setting `block_for` to `0` will cause queue workers to block indefinitely until a job is available. This will also prevent signals such as `SIGTERM` from being handled until the next job has been processed.

Other Driver Prerequisites

The following dependencies are needed for the listed queue drivers. These dependencies may be installed via the Composer package manager:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` or phpredis PHP extension

Creating Jobs

Generating Job Classes

By default, all of the queueable jobs for your application are stored in the `app/Jobs` directory. If the `app/Jobs` directory doesn't exist, it will be created when you run the `make:job` Artisan command:

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

The generated class will implement the `Illuminate\Contracts\Queue\ShouldQueue` interface, indicating to Laravel that the job should be pushed onto the queue to run asynchronously.

Job stubs may be customized using stub publishing.

Class Structure

Job classes are very simple, normally containing only a `handle` method that is invoked when the job is processed by the queue. To get started, let's take a look at an example job class. In this example, we'll pretend we manage a podcast publishing service and need to process the uploaded podcast files before they are published:

```
<?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;
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...
    }
}
```

In this example, note that we were able to pass an Eloquent model directly into the queued job's constructor. Because of the `SerializesModels` trait that the job is using, Eloquent models and their loaded relationships will be gracefully serialized and unserialized when the job is processing.

If your queued job accepts an Eloquent model in its constructor, only the identifier for the model will be serialized onto the queue. When the job is actually handled, the queue system will automatically re-retrieve the full model instance and its loaded relationships from the database. This approach to model serialization allows for much smaller job payloads to be sent to your queue driver.

`handle` Method Dependency Injection

The `handle` method is invoked when the job is processed by the queue. Note that we are able to type-hint dependencies on the `handle` method of the job. The Laravel service container automatically injects these dependencies.

If you would like to take total control over how the container injects dependencies into the `handle` method, you may use the container's `bindMethod` method. The `bindMethod` method accepts a callback which receives the job and the container. Within the callback, you are free to invoke the `handle` method however you wish. Typically, you should call this method from the `boot` method of your `App\Providers\AppServiceProvider` service provider:

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


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

Binary data, such as raw image contents, should be passed through the `base64_encode` function before being passed to a queued job. Otherwise, the

job may not properly serialize to JSON when being placed on the queue.

Handling Relationships

Because loaded relationships also get serialized, the serialized job string can sometimes become quite large. To prevent relations from being serialized, you can call the `withoutRelations` method on the model when setting a property value. This method will return an instance of the model without its loaded relationships:

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

Unique Jobs

Unique jobs require a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks. In addition, unique job constraints do not apply to jobs within batches.

Sometimes, you may want to ensure that only one instance of a specific job is on the queue at any point in time. You may do so by implementing the `ShouldBeUnique` interface on your job class. This interface does not require you to define any additional methods on your class:

```
<?php

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

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

In the example above, the `UpdateSearchIndex` job is unique. So, the job will not be dispatched if another instance of the job is already on the queue and has not finished processing.

In certain cases, you may want to define a specific "key" that makes the job unique or you may want to specify a timeout beyond which the job no longer stays unique. To accomplish this, you may define `uniqueId` and `uniqueFor` properties or methods on your job class:

```
<?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;
    }
}
```

In the example above, the `UpdateSearchIndex` job is unique by a product ID. So, any new dispatches of the job with the same product ID will be ignored until the existing job has completed processing. In addition, if the existing job is not processed within one hour, the unique lock will be released and another job with the same unique key can be dispatched to the queue.

Keeping Jobs Unique Until Processing Begins

By default, unique jobs are "unlocked" after a job completes processing or fails all of its retry attempts. However, there may be situations where you would like your job to unlock immediately before it is processed. To accomplish this, your job should implement the `ShouldBeUniqueUntilProcessing` contract instead of the `ShouldBeUnique` contract:

```php
<?php

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

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

Unique Job Locks

Behind the scenes, when a `ShouldBeUnique` job is dispatched, Laravel attempts to acquire a lock with the `uniqueId` key. If the lock is not acquired, the job is not dispatched. This lock is released when the job completes processing or fails all of its retry attempts. By default, Laravel will use the default cache driver to obtain this lock. However, if you wish to use another driver for acquiring the lock, you may define a `uniqueVia` method that returns the cache driver that should be used:

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

If you only need to limit the concurrent processing of a job, use the `WithoutOverlapping` job middleware instead.

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves. For example, consider the following `handle` method which leverages Laravel's Redis rate limiting features to allow only one job to process every five seconds:

```
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);
```

```
        });
    }
```

While this code is valid, the implementation of the `handle` method becomes noisy since it is cluttered with Redis rate limiting logic. In addition, this rate limiting logic must be duplicated for any other jobs that we want to rate limit.

Instead of rate limiting in the handle method, we could define a job middleware that handles rate limiting. Laravel does not have a default location for job middleware, so you are welcome to place job middleware anywhere in your application. In this example, we will place the middleware in an `app/Jobs/Middleware` directory:

```
<?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);
                });
        }
    }
```

As you can see, like route middleware, job middleware receive the job being processed and a callback that should be invoked to continue processing the job.

After creating job middleware, they may be attached to a job by returning them from the job's `middleware` method. This method does not exist on jobs scaffolded by the `make:job` Artisan command, so you will need to manually add it to your job class:

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

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

Rate Limiting

Although we just demonstrated how to write your own rate limiting job middleware, Laravel actually includes a rate limiting middleware that you may utilize to rate limit jobs. Like route rate limiters, job rate limiters are defined using the `RateLimiter` facade's `for` method.

For example, you may wish to allow users to backup their data once per hour while imposing no such limit on premium customers. To accomplish this, you may define a `RateLimiter` in the `boot` method of your `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);
    });
}
```

In the example above, we defined an hourly rate limit; however, you may easily define a rate limit based on minutes using the `perMinute` method. In addition, you may pass any value you wish to the `by` method of the rate limit; however, this value is most often used to segment rate limits by customer:

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

Once you have defined your rate limit, you may attach the rate limiter to your backup job using the `Illuminate\Queue\Middleware\RateLimited` middleware. Each time the job exceeds the rate limit, this middleware will release the job back to the queue with an appropriate delay based on the rate limit duration.

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

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

Releasing a rate limited job back onto the queue will still increment the job's total number of `attempts`. You may wish to tune your `tries` and `maxExceptions` properties on your job class accordingly. Or, you may wish to use the `retryUntil` method to define the amount of time until the job should no longer be attempted.

If you are using Redis, you may use the `Illuminate\Queue\Middleware\RateLimitedWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic rate limiting middleware.

Preventing Job Overlaps

Laravel includes an `Illuminate\Queue\Middleware\WithoutOverlapping` middleware that allows you to prevent job overlaps based on an arbitrary key. This can be helpful when a queued job is modifying a resource that should only be modified by one job at a time.

For example, let's imagine you have a queued job that updates a user's credit score and you want to prevent credit score update job overlaps for the same user ID. To accomplish this, you can return the `WithoutOverlapping` middleware from your job's `middleware` method:

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

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

Any overlapping jobs will be released back to the queue. You may also specify the number of seconds that must elapse before the released job will be attempted again:

```
/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [(new WithoutOverlapping($this->order->id))->releaseAfter(60)];
}
```

If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the `dontRelease` method:

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

The `WithoutOverlapping` middleware requires a cache driver that supports locks. Currently, the `memcached`, `redis`, `dynamodb`, `database`, `file`, and `array` cache drivers support atomic locks.

Throttling Exceptions

Laravel includes a `Illuminate\Queue\Middleware\ThrottlesExceptions` middleware that allows you to throttle exceptions. Once the job throws a given number of exceptions, all further attempts to execute the job are delayed until a specified time interval lapses. This middleware is particularly useful for jobs that interact with third-party services that are unstable.

For example, let's imagine a queued job that interacts with an third-party API that begins throwing exceptions. To throttle exceptions, you can return the `ThrottlesExceptions` middleware from your job's `middleware` method. Typically, this middleware should be paired with a job that implements time based attempts:

```php
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(30);
}
```

The first constructor argument accepted by the middleware is the number of exceptions the job can throw before being throttled, while the second constructor argument is the number of minutes that should elapse before the job is attempted again once it has been throttled. In the code example above, if the job throws 10 exceptions within 5 minutes, we will wait 5 minutes before attempting the job again.

When a job throws an exception but the exception threshold has not yet been reached, the job will typically be retried immediately. However, you may specify the number of seconds such a job should be delayed by calling the `backoff` method when attaching the middleware to the job:

```
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)];
}
```

Internally, this middleware uses Laravel's cache system to implement rate limiting, and the job's class name is utilized as the cache "key". You may override this key by calling the `by` method when attaching the middleware to your job. This may be useful if you have multiple jobs interacting with the same third-party service and you would like them to share a common throttling "bucket":

```
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')];
}
```

If you are using Redis, you may use the `Illuminate\Queue\Middleware\ThrottlesExceptionsWithRedis` middleware, which is fine-tuned for Redis and more efficient than the basic exception throttling middleware.

Dispatching Jobs

Once you have written your job class, you may dispatch it using the `dispatch` method on the job itself. The arguments passed to the `dispatch` method will be given to the job's constructor:

```
<?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);
```

```
    }
}
```

If you would like to conditionally dispatch a job, you may use the `dispatchIf` and `dispatchUnless` methods:

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

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

Delayed Dispatching

If you would like to specify that a job should not be immediately available for processing by a queue worker, you may use the `delay` method when dispatching the job. For example, let's specify that a job should not be available for processing until 10 minutes after it has been dispatched:

```
<?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));
    }
}
```

The Amazon SQS queue service has a maximum delay time of 15 minutes.

Dispatching After The Response Is Sent To Browser

Alternatively, the `dispatchAfterResponse` method delays dispatching a job until after the HTTP response is sent to the user's browser. This will still allow the user to begin using the application even though a queued job is still executing. This should typically only be used for jobs that take about a second, such as sending an email. Since they are processed within the current HTTP request, jobs dispatched in this fashion do not require a queue worker to be running in order for them to be processed:

```
use App\Jobs\SendNotification;

SendNotification::dispatchAfterResponse();
```

You may also `dispatch` a closure and chain the `afterResponse` method onto the `dispatch` helper to execute a closure after the HTTP response has been sent to the

browser:

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

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

Synchronous Dispatching

If you would like to dispatch a job immediately (synchronously), you may use the `dispatchSync` method. When using this method, the job will not be queued and will be executed immediately within the current process:

```
<?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);
    }
}
```

Jobs & Database Transactions

While it is perfectly fine to dispatch jobs within database transactions, you should take special care to ensure that your job will actually be able to execute successfully. When dispatching a job within a transaction, it is possible that the job will be processed by a worker before the transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database.

Thankfully, Laravel provides several methods of working around this problem. First, you may set the `after_commit` connection option in your queue connection's configuration array:

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

When the `after_commit` option is `true`, you may dispatch jobs within database transactions; however, Laravel will wait until all open database transactions have been committed before actually dispatching the job. Of course, if no database transactions are currently open, the job will be dispatched immediately.

If a transaction is rolled back due to an exception that occurs during the transaction, the dispatched jobs that were dispatched during that transaction

will be discarded.

Setting the `after_commit` configuration option to `true` will also cause any queued event listeners, mailables, notifications, and broadcast events to be dispatched after all open database transactions have been committed.

Specifying Commit Dispatch Behavior Inline

If you do not set the `after_commit` queue connection configuration option to `true`, you may still indicate that a specific job should be dispatched after all open database transactions have been committed. To accomplish this, you may chain the `afterCommit` method onto your dispatch operation:

```
use App\Jobs\ProcessPodcast;

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

Likewise, if the `after_commit` configuration option is set to `true`, you may indicate that a specific job should be dispatched immediately without waiting for any open database transactions to commit:

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

Job Chaining

Job chaining allows you to specify a list of queued jobs that should be run in sequence after the primary job has executed successfully. If one job in the sequence fails, the rest of the jobs will not be run. To execute a queued job chain, you may use the `chain` method provided by the `Bus` facade. Laravel's command bus is a lower level component that queued job dispatching is built on top of:

```
use App\Jobs\OptimizePodcast;
use App\Jobs\ProcessPodcast;
use App\Jobs\ReleasePodcast;
use Illuminate\Support\Facades\Bus;

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

In addition to chaining job class instances, you may also chain closures:

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

Deleting jobs using the `$this->delete()` method within the job will not prevent chained jobs from being processed. The chain will only stop executing if a job

in the chain fails.

Chain Connection & Queue

If you would like to specify the connection and queue that should be used for the chained jobs, you may use the `onConnection` and `onQueue` methods. These methods specify the queue connection and queue name that should be used unless the queued job is explicitly assigned a different connection / queue:

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

Chain Failures

When chaining jobs, you may use the `catch` method to specify a closure that should be invoked if a job within the chain fails. The given callback will receive the `Throwable` instance that caused the job failure:

```
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();
```

Customizing The Queue & Connection

Dispatching To A Particular Queue

By pushing jobs to different queues, you may "categorize" your queued jobs and even prioritize how many workers you assign to various queues. Keep in mind, this does not push jobs to different queue "connections" as defined by your queue configuration file, but only to specific queues within a single connection. To specify the queue, use the `onQueue` method when dispatching the job:

```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(...);

        // Create podcast...

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

Alternatively, you may specify the job's queue by calling the `onQueue` method within the job's constructor:

```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');
    }
}
```

Dispatching To A Particular Connection

If your application interacts with multiple queue connections, you may specify which connection to push a job to using the `onConnection` method:

```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(...);

        // Create podcast...

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

You may chain the `onConnection` and `onQueue` methods together to specify the connection and the queue for a job:

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

Alternatively, you may specify the job's connection by calling the `onConnection` method within the job's constructor:

```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->onConnection('sqs');
    }
}
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

If one of your queued jobs is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a job may be attempted.

One approach to specifying the maximum number of times a job may be attempted is via the `--tries` switch on the Artisan command line. This will apply

to all jobs processed by the worker unless the job being processed specifies a more specific number of times it may be attempted:

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

If a job exceeds its maximum number of attempts, it will be considered a "failed" job. For more information on handling failed jobs, consult the failed job documentation.

You may take a more granular approach by defining the maximum number of times a job may be attempted on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the `--tries` value provided on the command line:

```
<?php

namespace App\Jobs;

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

Time Based Attempts

As an alternative to defining how many times a job may be attempted before it fails, you may define a time at which the job should no longer be attempted. This allows a job to be attempted any number of times within a given time frame. To

define the time at which a job should no longer be attempted, add a `retryUntil` method to your job class. This method should return a `DateTime` instance:

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

You may also define a `tries` property or `retryUntil` method on your queued event listeners.

Max Exceptions

Sometimes you may wish to specify that a job may be attempted many times, but should fail if the retries are triggered by a given number of unhandled exceptions (as opposed to being released by the `release` method directly). To accomplish this, you may define a `maxExceptions` property on your job class:

```
<?php

namespace App\Jobs;

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

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

In this example, the job is released for ten seconds if the application is unable to obtain a Redis lock and will continue to be retried up to 25 times. However, the job will fail if three unhandled exceptions are thrown by the job.

Timeout

The `pcntl` PHP extension must be installed in order to specify job timeouts.

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server.

The maximum number of seconds that jobs can run may be specified using the `--timeout` switch on the Artisan command line:

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

If the job exceeds its maximum attempts by continually timing out, it will be marked as failed.

You may also define the maximum number of seconds a job should be allowed to run on the job class itself. If the timeout is specified on the job, it will take precedence over any timeout specified on the command line:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
```

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

Sometimes, IO blocking processes such as sockets or outgoing HTTP connections may not respect your specified timeout. Therefore, when using these features, you should always attempt to specify a timeout using their APIs as well. For example, when using Guzzle, you should always specify a connection and request timeout value.

Error Handling

If an exception is thrown while the job is being processed, the job will automatically be released back onto the queue so it may be attempted again. The job will continue to be released until it has been attempted the maximum number of times allowed by your application. The maximum number of attempts is defined by the `--tries` switch used on the `queue:work` Artisan command. Alternatively, the maximum number of attempts may be defined on the job class itself. More information on running the queue worker can be found below.

Manually Releasing A Job

Sometimes you may wish to manually release a job back onto the queue so that it can be attempted again at a later time. You may accomplish this by calling the `release` method:

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

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

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

By default, the `release` method will release the job back onto the queue for immediate processing. However, by passing an integer to the `release` method you may instruct the queue to not make the job available for processing until a given number of seconds has elapsed:

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

Manually Failing A Job

Occasionally you may need to manually mark a job as "failed". To do so, you may call the `fail` method:

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

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

If you would like to mark your job as failed because of an exception that you have caught, you may pass the exception to the `fail` method:

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

For more information on failed jobs, check out the documentation on dealing with job failures.

Job Batching

Laravel's job batching feature allows you to easily execute a batch of jobs and then perform some action when the batch of jobs has completed executing. Before getting started, you should create a database migration to build a table to contain meta information about your job batches, such as their completion percentage. This migration may be generated using the `queue:batches-table` Artisan command:

```
php artisan queue:batches-table

php artisan migrate
```

Defining Batchable Jobs

To define a batchable job, you should create a queueable job as normal; however, you should add the `Illuminate\Bus\Batchable` trait to the job class. This trait provides access to a `batch` method which may be used to retrieve the current batch that the job is executing within:

```
<?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...
    }
}
```

Dispatching Batches

To dispatch a batch of jobs, you should use the `batch` method of the `Bus` facade. Of course, batching is primarily useful when combined with completion callbacks. So, you may use the `then`, `catch`, and `finally` methods to define completion callbacks for the batch. Each of these callbacks will receive an

`Illuminate\Bus\Batch` instance when they are invoked. In this example, we will imagine we are queueing a batch of jobs that each process a given number of rows from a CSV file:

```
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;
```

The batch's ID, which may be accessed via the `$batch->id` property, may be used to query the Laravel command bus for information about the batch after it has been dispatched.

Naming Batches

Some tools such as Laravel Horizon and Laravel Telescope may provide more user-friendly debug information for batches if batches are named. To assign an arbitrary name to a batch, you may call the `name` method while defining the batch:

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

Batch Connection & Queue

If you would like to specify the connection and queue that should be used for the batched jobs, you may use the `onConnection` and `onQueue` methods. All batched jobs must execute within the same connection and queue:

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

Chains Within Batches

You may define a set of chained jobs within a batch by placing the chained jobs within an array. For example, we may execute two job chains in parallel and execute a callback when both job chains have finished processing:

```
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();
```

Adding Jobs To Batches

Sometimes it may be useful to add additional jobs to a batch from within a batched job. This pattern can be useful when you need to batch thousands of jobs which may take too long to dispatch during a web request. So, instead, you may wish to dispatch an initial batch of "loader" jobs that hydrate the batch with even more jobs:

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

In this example, we will use the `LoadImportBatch` job to hydrate the batch with additional jobs. To accomplish this, we may use the `add` method on the batch instance that may be accessed via the job's `batch` method:

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

You may only add jobs to a batch from within a job that belongs to the same batch.

Inspecting Batches

The `Illuminate\Bus\Batch` instance that is provided to batch completion callbacks has a variety of properties and methods to assist you in interacting with and inspecting a given batch of jobs:

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

// 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();
```

Returning Batches From Routes

All `Illuminate\Bus\Batch` instances are JSON serializable, meaning you can return them directly from one of your application's routes to retrieve a JSON payload containing information about the batch, including its completion progress. This makes it convenient to display information about the batch's completion progress in your application's UI.

To retrieve a batch by its ID, you may use the `Bus` facade's `findBatch` method:

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

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

Cancelling Batches

Sometimes you may need to cancel a given batch's execution. This can be accomplished by calling the `cancel` method on the `Illuminate\Bus\Batch` instance:

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

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

As you may have noticed in previous examples, batched jobs should typically check to see if the batch has been cancelled at the beginning of their `handle` method:

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

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

    // Continue processing...
}
```

Batch Failures

When a batched job fails, the `catch` callback (if assigned) will be invoked. This callback is only invoked for the first job that fails within the batch.

Allowing Failures

When a job within a batch fails, Laravel will automatically mark the batch as "cancelled". If you wish, you may disable this behavior so that a job failure does not automatically mark the batch as cancelled. This may be accomplished by calling the `allowFailures` method while dispatching the batch:

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

Retrying Failed Batch Jobs

For convenience, Laravel provides a `queue:retry-batch` Artisan command that allows you to easily retry all of the failed jobs for a given batch. The `queue:retry-batch` command accepts the UUID of the batch whose failed jobs should be retried:

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

Pruning Batches

Without pruning, the `job_batches` table can accumulate records very quickly. To mitigate this, you should schedule the `queue:prune-batches` Artisan command to run daily:

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

By default, all finished batches that are more than 24 hours old will be pruned. You may use the `hours` option when calling the command to determine how long to retain batch data. For example, the following command will delete all batches that finished over 48 hours ago:

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

Queueing Closures

Instead of dispatching a job class to the queue, you may also dispatch a closure. This is great for quick, simple tasks that need to be executed outside of the current request cycle. When dispatching closures to the queue, the closure's code content is cryptographically signed so that it can not be modified in transit:

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

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

Using the `catch` method, you may provide a closure that should be executed if the queued closure fails to complete successfully after exhausting all of your queue's configured retry attempts:

```
use Throwable;

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

Running The Queue Worker

The `queue:work` Command

Laravel includes an Artisan command that will start a queue worker and process new jobs as they are pushed onto the queue. You may run the worker using the `queue:work` Artisan command. Note that once the `queue:work` command has started, it will continue to run until it is manually stopped or you close your terminal:

```
php artisan queue:work
```

To keep the `queue:work` process running permanently in the background, you should use a process monitor such as Supervisor to ensure that the queue worker does not stop running.

Remember, queue workers, are long-lived processes and store the booted application state in memory. As a result, they will not notice changes in your code

base after they have been started. So, during your deployment process, be sure to restart your queue workers. In addition, remember that any static state created or modified by your application will not be automatically reset between jobs.

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker when you want to reload your updated code or reset the application state; however, this command is significantly less efficient than the `queue:work` command:

```
php artisan queue:listen
```

Running Multiple Queue Workers

To assign multiple workers to a queue and process jobs concurrently, you should simply start multiple `queue:work` processes. This can either be done locally via multiple tabs in your terminal or in production using your process manager's configuration settings. When using Supervisor, you may use the `numprocs` configuration value.

Specifying The Connection & Queue

You may also specify which queue connection the worker should utilize. The connection name passed to the `work` command should correspond to one of the connections defined in your `config/queue.php` configuration file:

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

You may customize your queue worker even further by only processing particular queues for a given connection. For example, if all of your emails are processed in an `emails` queue on your `redis` queue connection, you may issue the following command to start a worker that only processes that queue:

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

Processing A Specified Number Of Jobs

The `--once` option may be used to instruct the worker to only process a single job from the queue:

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

The `--max-jobs` option may be used to instruct the worker to process the given number of jobs and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after processing a given number of jobs, releasing any memory they may have accumulated:

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

Processing All Queued Jobs & Then Exiting

The `--stop-when-empty` option may be used to instruct the worker to process all jobs and then exit gracefully. This option can be useful when processing Laravel queues within a Docker container if you wish to shutdown the container after the queue is empty:

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

Processing Jobs For A Given Number Of Seconds

The `--max-time` option may be used to instruct the worker to process jobs for the given number of seconds and then exit. This option may be useful when combined with Supervisor so that your workers are automatically restarted after

processing jobs for a given amount of time, releasing any memory they may have accumulated:

```
// Process jobs for one hour and then exit...
php artisan queue:work --max-time=3600
```

Worker Sleep Duration

When jobs are available on the queue, the worker will keep processing jobs with no delay in between them. However, the `sleep` option determines how many seconds the worker will "sleep" if there are no new jobs available. While sleeping, the worker will not process any new jobs - the jobs will be processed after the worker wakes up again.

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

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should release any heavy resources after each job completes. For example, if you are doing image manipulation with the GD library, you should free the memory with `imagedestroy` when you are done processing the image.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` configuration file, you may set the default `queue` for your `redis` connection to `low`. However, occasionally you may wish to push a job to a `high` priority queue like so:

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

To start a worker that verifies that all of the `high` queue jobs are processed before continuing to any jobs on the `low` queue, pass a comma-delimited list of queue names to the `work` command:

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

Queue Workers & Deployment

Since queue workers are long-lived processes, they will not notice changes to your code without being restarted. So, the simplest way to deploy an application using queue workers is to restart the workers during your deployment process. You may gracefully restart all of the workers by issuing the `queue:restart` command:

```
php artisan queue:restart
```

This command will instruct all queue workers to gracefully exit after they finish processing their current job so that no existing jobs are lost. Since the queue workers will exit when the `queue:restart` command is executed, you should be running a process manager such as Supervisor to automatically restart the queue workers.

The queue uses the cache to store restart signals, so you should verify that a cache driver is properly configured for your application before using this feature.

Job Expirations & Timeouts

Job Expiration

In your `config/queue.php` configuration file, each queue connection defines a `retry_after` option. This option specifies how many seconds the queue connection should wait before retrying a job that is being processed. For example, if the value of `retry_after` is set to `90`, the job will be released back onto the queue if it has been processing for 90 seconds without being released or deleted. Typically, you should set the `retry_after` value to the maximum number of seconds your jobs should reasonably take to complete processing.

The only queue connection which does not contain a `retry_after` value is Amazon SQS. SQS will retry the job based on the Default Visibility Timeout which is managed within the AWS console.

Worker Timeouts

The `queue:work` Artisan command exposes a `--timeout` option. If a job is processing for longer than the number of seconds specified by the timeout value, the worker processing the job will exit with an error. Typically, the worker will be restarted automatically by a process manager configured on your server:

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

The `retry_after` configuration option and the `--timeout` CLI option are different, but work together to ensure that jobs are not lost and that jobs are only successfully processed once.

The `--timeout` value should always be at least several seconds shorter than your `retry_after` configuration value. This will ensure that a worker processing a frozen job is always terminated before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

Supervisor Configuration

In production, you need a way to keep your `queue:work` processes running. A `queue:work` process may stop running for a variety of reasons, such as an exceeded worker timeout or the execution of the `queue:restart` command.

For this reason, you need to configure a process monitor that can detect when your `queue:work` processes exit and automatically restart them. In addition, process monitors can allow you to specify how many `queue:work` processes you would like to run concurrently. Supervisor is a process monitor commonly used in Linux environments and we will discuss how to configure it in the following documentation.

Installing Supervisor

Supervisor is a process monitor for the Linux operating system, and will automatically restart your `queue:work` processes if they fail. To install Supervisor on Ubuntu, you may use the following command:

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

If configuring and managing Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your production Laravel projects.

Configuring Supervisor

Supervisor configuration files are typically stored in the `/etc/supervisor/conf.d` directory. Within this directory, you may create any number of configuration files that instruct supervisor how your processes should be monitored. For example, let's create a `laravel-worker.conf` file that starts and monitors `queue:work` processes:

```
[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
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
```

In this example, the `numprocs` directive will instruct Supervisor to run eight `queue:work` processes and monitor all of them, automatically restarting them if

they fail. You should change the `command` directive of the configuration to reflect your desired queue connection and worker options.

You should ensure that the value of `stopwaitsecs` is greater than the number of seconds consumed by your longest running job. Otherwise, Supervisor may kill the job before it is finished processing.

Starting Supervisor

Once the configuration file has been created, you may update the Supervisor configuration and start the processes using the following commands:

```
sudo supervisorctl reread

sudo supervisorctl update

sudo supervisorctl start laravel-worker:*
```

For more information on Supervisor, consult the Supervisor documentation.

Dealing With Failed Jobs

Sometimes your queued jobs will fail. Don't worry, things don't always go as planned! Laravel includes a convenient way to specify the maximum number of times a job should be attempted. After a job has exceeded this number of attempts, it will be inserted into the `failed_jobs` database table. Of course, we will

need to create that table if it does not already exist. To create a migration for the `failed_jobs` table, you may use the `queue:failed-table` command:

```
php artisan queue:failed-table

php artisan migrate
```

When running a queue worker process, you may specify the maximum number of times a job should be attempted using the `--tries` switch on the `queue:work` command. If you do not specify a value for the `--tries` option, jobs will only be attempted once or as many times as specified by the job class' `$tries` property:

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

Using the `--backoff` option, you may specify how many seconds Laravel should wait before retrying a job that has encountered an exception. By default, a job is immediately released back onto the queue so that it may be attempted again:

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

If you would like to configure how many seconds Laravel should wait before retrying a job that has encountered an exception on a per-job basis, you may do so by defining a `backoff` property on your job class:

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

If you require more complex logic for determining the job's backoff time, you may define a `backoff` method on your job class:

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

You may easily configure "exponential" backoffs by returning an array of backoff values from the `backoff` method. In this example, the retry delay will be 1 second for the first retry, 5 seconds for the second retry, and 10 seconds for the third retry:

```
/**
* Calculate the number of seconds to wait before retrying the job.
*
* @return array
*/
public function backoff()
{
    return [1, 5, 10];
}
```

Cleaning Up After Failed Jobs

When a particular job fails, you may want to send an alert to your users or revert any actions that were partially completed by the job. To accomplish this, you may define a `failed` method on your job class. The `Throwable` instance that caused the job to fail will be passed to the `failed` method:

```php
<?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
     */
    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...
    }

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

Retrying Failed Jobs

To view all of the failed jobs that have been inserted into your `failed_jobs` database table, you may use the `queue:failed` Artisan command:

```
php artisan queue:failed
```

The `queue:failed` command will list the job ID, connection, queue, failure time, and other information about the job. The job ID may be used to retry the failed job. For instance, to retry a failed job that has an ID of `5`, issue the following command:

```
php artisan queue:retry 5
```

If necessary, you may pass multiple IDs or an ID range (when using numeric IDs) to the command:

```
php artisan queue:retry 5 6 7 8 9 10

php artisan queue:retry --range=5-10
```

To retry all of your failed jobs, execute the `queue:retry` command and pass `all` as the ID:

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

If you would like to delete a failed job, you may use the `queue:forget` command:

```
php artisan queue:forget 5
```

When using Horizon, you should use the `horizon:forget` command to delete a failed job instead of the `queue:forget` command.

To delete all of your failed jobs from the `failed_jobs` table, you may use the `queue:flush` command:

```
php artisan queue:flush
```

Ignoring Missing Models

When injecting an Eloquent model into a job, the model is automatically serialized before being placed on the queue and re-retrieved from the database when the job is processed. However, if the model has been deleted while the job was waiting to be processed by a worker, your job may fail with a `ModelNotFoundException`.

For convenience, you may choose to automatically delete jobs with missing models by setting your job's `deleteWhenMissingModels` property to `true`. When this property is set to `true`, Laravel will quietly discard the job without raising an exception:

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

Failed Job Events

If you would like to register an event listener that will be invoked when a job fails, you may use the `Queue` facade's `failing` method. For example, we may attach a closure to this event from the `boot` method of the `AppServiceProvider` that is included with Laravel:

```
<?php

namespace App\Providers;

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

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

Clearing Jobs From Queues

When using Horizon, you should use the `horizon:clear` command to clear jobs from the queue instead of the `queue:clear` command.

If you would like to delete all jobs from the default queue of the default connection, you may do so using the `queue:clear` Artisan command:

```
php artisan queue:clear
```

You may also provide the `connection` argument and `queue` option to delete jobs from a specific connection and queue:

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

Clearing jobs from queues is only available for the SQS, Redis, and database queue drivers. In addition, the SQS message deletion process takes up to 60 seconds, so jobs sent to the SQS queue up to 60 seconds after you clear the queue might also be deleted.

Job Events

Using the `before` and `after` methods on the `Queue` facade, you may specify callbacks to be executed before or after a queued job is processed. These callbacks are a great opportunity to perform additional logging or increment statistics for a dashboard. Typically, you should call these methods from the `boot` method of a service provider. For example, we may use the `AppServiceProvider` that is included with 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()
```

```
            });
        }
    }
```

Using the `looping` method on the `Queue` facade, you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a closure to rollback any transactions that were left open by a previously failed job:

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

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

Highlights +

Resources +

Partners +

Ecosystem +

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.