Queues

# Introduction

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

Laravel queues provide a unified API across a variety of different queue backends, such as Beanstalk, Amazon SQS, Redis, or even a relational database. Queues allow you to defer the processing of a time consuming task, such as sending an email, until a later time. Deferring these time consuming tasks drastically speeds up web requests to your application.

The queue configuration file is stored in `config/queue.php`. In this file you will find connection configurations for each of the queue drivers that are included with the framework, which includes a database, Beanstalkd, Amazon SQS, Redis, and a synchronous driver that will execute jobs immediately (for local use). A `null` queue driver is also included which discards queued jobs.

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 option. This option defines a particular connection to a backend service 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:

```
// This job is sent to the default queue...
Job::dispatch();

// This job is sent to the "emails" queue...
Job::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:

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

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. You may generate a new queued job using the Artisan CLI:

```
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.

Class Structure

Job classes are very simple, normally containing only a `handle` method which is called 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
<?php

namespace App\Jobs;

use App\AudioProcessor;
use App\Podcast;
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;

    protected $podcast;

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

    /**
     * Execute the job.
     *
     * @param  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 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 from the database. It's all totally transparent to your application and prevents issues that can arise from serializing full Eloquent model instances.

The `handle` method is called 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 a service provider:

```php
use App\Jobs\ProcessPodcast;

$this->app->bindMethod(ProcessPodcast::class.'@handle', function ($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.

Job Middleware

Job middleware allow you 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:

```
/**
 * 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 structure 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 a `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 add it to your own job class definition:

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

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

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 Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

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

Delayed Dispatching

If you would like to delay the execution of a queued job, you may use the `delay` method when dispatching a 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 Illuminate\Http\Request;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

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

!

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

Synchronous Dispatching

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

```
<?php
```

```
namespace App\Http\Controllers;

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

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

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 `withChain` method on any of your dispatchable jobs:

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

! Deleting jobs using the `$this->delete()` method 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 default connection and queue that should be used for the chained jobs, you may use the `allOnConnection` and `allOnQueue` 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:

```
ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->dispatch()->allOnConnection('redis')->allOnQueue('podcasts');
```

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

namespace App\Http\Controllers;

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

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

Dispatching To A Particular Connection

If you are working with multiple queue connections, you may specify which connection to push a job to. To specify the connection, use the `onConnection` method when dispatching the job:

```php
<?php

namespace App\Http\Controllers;

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

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

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

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

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

Alternatively, you may specify the `connection` as a property on the job class:

```php
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The queue connection that should handle the job.
     *
     * @var string
     */
    public $connection = 'sqs';
}
```

Specifying Max Job Attempts / Timeout Values

Max Attempts

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

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

However, you may take a more granular approach by defining the maximum number of attempts on the job class itself. If the maximum number of attempts is specified on the job, it will take precedence over the value provided on the command line:

```php
<?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 timeout. 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 timeout, add a `retryUntil` method to your job class:

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

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

Timeout

The `timeout` feature is optimized for PHP 7.1+ and the `pcntl` PHP extension.

Likewise, 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
```

However, 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;
}
```

Rate Limiting

This feature requires that your application can interact with a Redis server.

If your application interacts with Redis, you may throttle your queued jobs by time or concurrency. This feature can be of assistance when your queued jobs are interacting with APIs that are also rate limited.

For example, using the `throttle` method, you may throttle a given type of job to only run 10 times every 60 seconds. If a lock can not be obtained, you should typically release the job back onto the queue so it can be retried later:

```
Redis::throttle('key')->allow(10)->every(60)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

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

In the example above, the `key` may be any string that uniquely identifies the type of job you would like to rate limit. For example, you may wish to construct the key based on the class name of the job and the IDs of the Eloquent models it operates on.

Releasing a throttled job back onto the queue will still increment the job's total number of `attempts`.

Alternatively, you may specify the maximum number of workers that may simultaneously process a given job. 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, using the `funnel` method, you may limit jobs of a given type to only be processed by one worker at a time:

```
Redis::funnel('key')->limit(1)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

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

When using rate limiting, the number of attempts your job will need to run successfully can be hard to determine. Therefore, it is useful to combine rate limiting with time based attempts.

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.

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:

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

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

When dispatching Closures to the queue, the Closure's code contents is cryptographically signed so it can not be modified in transit.

Running The Queue Worker

Laravel includes a queue worker that will 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.

Alternatively, you may run the `queue:listen` command. When using the `queue:listen` command, you don't have to manually restart the worker after your code is changed; however, this command is not as efficient as `queue:work`:

```
php artisan queue:listen
```

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 only that queue:

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

Processing A Single Job

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

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

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 working 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
```

Resource Considerations

Daemon queue workers do not "reboot" the framework before processing each job. Therefore, you should free 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.

Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your `config/queue.php` 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 pick up 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 "die" after they finish processing their current job so that no existing jobs are lost. Since the queue workers will die 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 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 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. The `--timeout` option specifies how long the Laravel queue master process will wait before killing off a child queue worker that is processing a job. Sometimes a child queue process can become "frozen" for various reasons, such as an external HTTP call that is not responding. The `--timeout` option removes frozen processes that have exceeded that specified time limit:

```
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 given job is always killed before the job is retried. If your `--timeout` option is longer than your `retry_after` configuration value, your jobs may be processed twice.

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 long (in 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
```

Supervisor Configuration

Installing Supervisor

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

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

If configuring Supervisor yourself sounds overwhelming, consider using Laravel Forge, which will automatically install and configure Supervisor for your 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 a `queue:work` process:

```
[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
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

In this example, the `numprocs` directive will instruct Supervisor to run 8 `queue:work` processes and monitor all of them, automatically restarting them if they fail. You should change the `queue:work sqs` portion of the `command` directive to reflect your desired queue connection.

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 amount of attempts, it will be inserted into the `failed_jobs` database table. 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
```

Then, when running your queue worker, you can 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:

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

In addition, you may specify how many seconds Laravel should wait before retrying a job that has failed using the `--delay` option. By default, a job is retried immediately:

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

If you would like to configure the failed job retry delay on a per-job basis, you may do so by defining a `retryAfter` property on your queued job class:

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

Cleaning Up After Failed Jobs

You may define a `failed` method directly on your job class, allowing you to perform job specific clean-up when a failure occurs. This is the perfect location to send an alert to your users or revert any actions performed by the job. The `Exception` that caused the job to fail will be passed to the `failed` method:

```php
<?php

namespace App\Jobs;

use App\AudioProcessor;
use App\Podcast;
use Exception;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Queue\SerializesModels;

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

    protected $podcast;

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

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

    /**
     * The job failed to process.
     *
     * @param  Exception  $exception
     * @return void
     */
    public function failed(Exception $exception)
    {
        // Send user notification of failure, etc...
    }
}
```

Failed Job Events

If you would like to register an event that will be called when a job fails, you may use the `Queue::failing` method. This event is a great opportunity to notify your team via email or Slack. For example, we may attach a callback to this event from 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\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
        });
    }
}
```

Retrying Failed Jobs

To view all of your 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, and failure time. 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
```

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
```

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

```
php artisan queue:flush
```

Ignoring Missing Models

When injecting an Eloquent model into a job, it is automatically serialized before being placed on the queue and restored 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`:

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

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 a service provider. For example, we may use the `AppServiceProvider` that is included with Laravel:

```
<?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:

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

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

Release Notes
Getting Started
Routing
Blade Templates
Authentication
Authorization
Artisan Console
Database
Eloquent ORM
Testing

Resources

Laracasts
Laravel News
Laracon
Laracon EU
Laracon AU
Jobs
Certification
Forums

Partners

Vehikl
Tighten Co.
Kirschbaum
Byte 5
64Robots
Cubet
DevSquad
Ideil
Cyber-Duck
ABOUT YOU
Become A Partner

Ecosystem

Vapor
Forge
Envoyer
Horizon
Lumen
Nova
Echo
Valet
Mix
Spark
Cashier
Homestead
Dusk
Passport
Scout
Socialite

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.