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 the 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. If **--tries=0** is provided to the **queue:work** command, the job will be retried indefinitely.

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:

```
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:

```
use DateTime;

public function retryUntil(): DateTime

{

    return now()->addMinutes(10);

}
```

****

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:

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

class ProcessPodcast implements ShouldQueue

{

    /**

     * The number of times the job may be attempted.

     *

     * @var int

     */

    public $tries = 25;
```

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


    /**
     * Execute the job.
     */
    public function handle(): void
    {
        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.

****

Note: The **pcntl PHP extension** must be installed in order to specify job timeouts.

****

Timeout

Often, you know roughly how long you expect your queued jobs to take. For this reason, Laravel allows you to specify a "timeout" value. By default, the timeout value is 60 seconds. 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:

```
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**.

****

Failing On Timeout

If you would like to indicate that a job should be marked as failed on timeout, you may define the **$failOnTimeout** property on the job class:

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

****

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.
 */
public function handle(): void
{
    // …
    $this->release();
}
```

****

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

```
$this->release(10);
$this->release(now()->addSeconds(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:

```
public function handle(): void
{
    // ...
    $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. Or, for convenience, you may pass a string error message which will be converted to an exception for you:

```
$this->fail($exception);
$this->fail('Something went wrong.');
```

****