Laravel Queues

Introduction

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.

Driver Notes and Prerequisites

Database

In order to use the database queue driver, you will need a database table to hold the jobs. Typically, this is included in Laravel's default 0001_01_01_000002_create_jobs_table.php database migration; however, if your application does not contain this migration, you may use the make:queue-table Artisan command to create it.

Redis

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

The serializer and compression Redis options are not supported by the redis queue driver.

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' => env('REDIS_QUEUE_CONNECTION', 'default'),
   'queue' => env('REDIS_QUEUE', '{default}'),
   'retry_after' => env('REDIS_QUEUE_RETRY_AFTER', 90),
   'block_for' => null,
   'after_commit' => false,
],
```

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' => env('REDIS_QUEUE_CONNECTION', 'default'),
   'queue' => env('REDIS_QUEUE', 'default'),
```

```
    'retry_after' => env('REDIS_QUEUE_RETRY_AFTER', 90),
    'block_for' => 5,
    'after_commit' => false,
],
```

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 ~5.0
- Redis: predis/predis ~2.0 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.

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.

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.

Queued Relationships

Because all loaded Eloquent model relationships also get serialized when a job is queued, the serialized job string can sometimes become quite large. Furthermore, when a job is deserialized and model relationships are re-retrieved from the database, they will be retrieved in their entirety. Any previous relationship constraints that were applied before the model was serialized during the job queueing process will not be applied when the job is deserialized. Therefore, if you wish to work with a subset of a given relationship, you should re-constrain that relationship within your queued job.

Or, 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.

If a job receives a collection or array of Eloquent models instead of a single model, the models within that collection will not have their relationships restored when the job is deserialized and executed. This is to prevent excessive resource usage on jobs that deal with large numbers of models.

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.

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.

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.

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.

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

Encrypted Jobs

Laravel allows you to ensure the privacy and integrity of a job's data via encryption. To get started, simply add the ShouldBeEncrypted interface to the job class. Once this interface has been added to the class, Laravel will automatically encrypt your job before pushing it onto a queue.

Job Middleware

Job middleware allow you to wrap custom logic around the execution of queued jobs, reducing boilerplate in the jobs themselves.
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.
Job middleware can also be assigned to queueable event listeners, mailables, and notifications.

Preventing Job Overlaps

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.
Any overlapping jobs of the same type 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.
If you wish to immediately delete any overlapping jobs so that they will not be retried, you may use the dontRelease method.

Sharing Lock Keys Across Job Classes

By default, the WithoutOverlapping middleware will only prevent overlapping jobs of the same class. So, although two different job classes may use the same lock key, they will not be prevented from overlapping.

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 parent transaction has committed. When this happens, any updates you have made to models or database records during the database transaction(s) may not yet be reflected in the database. In addition, any models or database records created within the transaction(s) 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 the open parent 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 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.

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 and 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.

Adding Jobs to the Chain

Occasionally, you may need to prepend or append a job to an existing job chain from within another job in that chain. You may accomplish this using the prependToChain and appendToChain methods.4

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();
```

Since chain callbacks are serialized and executed at a later time by the Laravel queue, you should not use the $this variable within chain callbacks.

Customizing the Queue and 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.

Specifying Max Job Attempts / Timeout Values

Max Attempts

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

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;
/**
 * Determine the time at which the job should timeout.
 */
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.

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.

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.

The pcntl PHP extension must be installed in order to specify job timeouts. In addition, a job's "timeout" value should always be less than its "retry after" value. Otherwise, the job may be re-attempted before it has actually finished executing or timed out.

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.

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 which will contain meta information about your job batches, such as their completion percentage. This migration may be generated using the make:queue-batches-table Artisan command.

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
namespace App\Jobs;
use Illuminate\Bus\Batchable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Queue\Queueable;
class ImportCsv implements ShouldQueue
{
    use Batchable, Queueable;
    /**
     * Execute the job.
     */
    public function handle(): void
    {
        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),
])->before(function (Batch $batch) {
    // The batch has been created but no jobs have been added...
})->progress(function (Batch $batch) {
    // A single job has completed successfully...
})->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.

Since batch callbacks are serialized and executed at a later time by the Laravel queue, you should not use the $this variable within the callbacks. In addition, since batched jobs are wrapped within database transactions, database statements that trigger implicit commits should not be executed within the jobs.

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 and 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 and 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.

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.

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.

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.

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.

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:

use Illuminate\Support\Facades\Schedule;

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:

use Illuminate\Support\Facades\Schedule;

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

Sometimes, your jobs_batches table may accumulate batch records for batches that never completed successfully, such as batches where a job failed and that job was never retried successfully. You may instruct the queue:prune-batches command to prune these unfinished batch records using the unfinished option.

Storing Batches in DynamoDB

Laravel also provides support for storing batch meta information in DynamoDB instead of a relational database. However, you will need to manually create a DynamoDB table to store all of the batch records.

Typically, this table should be named job_batches, but you should name the table based on the value of the queue.batching.table configuration value within your application's queue configuration file.

DynamoDB Batch Table Configuration

The job_batches table should have a string primary partition key named application and a string primary sort key named id. The application portion of the key will contain your application's name as defined by the name configuration value within your application's app configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store job batches for multiple Laravel applications.

In addition, you may define ttl attribute for your table if you would like to take advantage of automatic batch pruning.

DynamoDB Configuration

Next, install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

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

Then, set the queue.batching.driver configuration option's value to dynamodb. In addition, you should define key, secret, and region configuration options within the batching configuration array. These options will be used to authenticate with AWS. When using the dynamodb driver, the queue.batching.database configuration option is unnecessary.

Pruning Batches in DynamoDB

When utilizing DynamoDB to store job batch information, the typical pruning commands used to prune batches stored in a relational database will not work. Instead, you may utilize DynamoDB's native TTL functionality to automatically remove records for old batches.

If you defined your DynamoDB table with a ttl attribute, you may define configuration parameters to instruct Laravel how to prune batch records. The queue.batching.ttl_attribute configuration value defines the name of the attribute holding the TTL, while the queue.batching.ttl configuration value defines the number of seconds after which a batch record can be removed from the DynamoDB table, relative to the last time the record was updated.

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.

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.

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.

You may include the -v flag when invoking the queue:work command if you would like the processed job IDs to be included in the command's output:

```
php artisan queue:work -v
```

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.

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; this command is significantly less efficient than the queue:work command.

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 and 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
```

By default, the queue:work command only processes jobs for the default queue on a given connection. However, 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 and 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 jobs. However, the sleep option determines how many seconds the worker will "sleep" if there are no jobs available. Of course, while sleeping, the worker will not process any new jobs:

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

Maintenance Mode and Queues

While your application is in maintenance mode, no queued jobs will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

To force your queue workers to process jobs even if maintenance mode is enabled, you may use --force option:

```
php artisan queue:work --force
```

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 and 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 and 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. 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:

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.

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 an asynchronous job has exceeded this number of attempts, it will be inserted into the failed_jobs database table. Synchronously dispatched jobs that fail are not stored in this table and their exceptions are immediately handled by the application.

A migration to create the failed_jobs table is typically already present in new Laravel applications. However, if your application does not contain a migration for this table, you may use the make:queue-failed-table command to create the migration.

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

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 ce7bb17c-cdd8-41f0-a8ec-7b4fef4e5ece, issue the following command:

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

If necessary, you may pass multiple IDs to the command:

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

You may also retry all of the failed jobs for a particular queue:

php artisan queue:retry --queue=name

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 91401d2c-0784-4f43-824c-34f94a33c24d

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.

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;
```

Pruning Failed Jobs

You may prune the records in your application's failed_jobs table by invoking the queue:prune-failed Artisan command:

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

By default, all the failed job records that are more than 24 hours old will be pruned. If you provide the --hours option to the command, only the failed job records that were inserted within the last N number of hours will be retained. For example, the following command will delete all the failed job records that were inserted more than 48 hours ago:

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

Storing Failed Jobs in DynamoDB

Laravel also provides support for storing your failed job records in DynamoDB instead of a relational database table. However, you must manually create a DynamoDB table to store all of the failed job records. Typically, this table should be named failed_jobs, but you should name the table based on the value of the queue.failed.table configuration value within your application's queue configuration file.

The failed_jobs table should have a string primary partition key named application and a string primary sort key named uuid. The application portion of the key will contain your application's name as defined by the name configuration value within your application's app configuration file. Since the application name is part of the DynamoDB table's key, you can use the same table to store failed jobs for multiple Laravel applications.

In addition, ensure that you install the AWS SDK so that your Laravel application can communicate with Amazon DynamoDB:

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

Next, set the queue.failed.driver configuration option's value to dynamodb. In addition, you should define key, secret, and region configuration options within the failed job configuration array. These options will be used to authenticate with AWS. When using the dynamodb driver, the queue.failed.database configuration option is unnecessary:

```
'failed' => [
    'driver' => env('QUEUE_FAILED_DRIVER', 'dynamodb'),
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
```

```
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'table' => 'failed_jobs',
],
```

Disabling Failed Job Storage

You may instruct Laravel to discard failed jobs without storing them by setting the queue.failed.driver configuration option's value to null. Typically, this may be accomplished via the QUEUE_FAILED_DRIVER environment variable:

```
QUEUE_FAILED_DRIVER=null
```

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.

Monitoring Your Queues

If your queue receives a sudden influx of jobs, it could become overwhelmed, leading to a long wait time for jobs to complete. If you wish, Laravel can alert you when your queue job count exceeds a specified threshold.

To get started, you should schedule the queue:monitor command to run every minute. The command accepts the names of the queues you wish to monitor as well as your desired job count threshold:

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

Scheduling this command alone is not enough to trigger a notification alerting you of the queue's overwhelmed status. When the command encounters a queue that has a job count exceeding your threshold, an Illuminate\Queue\Events\QueueBusy event will be dispatched. You may listen for this event within your application's AppServiceProvider in order to send a notification to you or your development team.

Testing

When testing code that dispatches jobs, you may wish to instruct Laravel to not actually execute the job itself, since the job's code can be tested directly and separately of the code that dispatches it. Of course,

to test the job itself, you may instantiate a job instance and invoke the handle method directly in your test.

You may use the Queue facade's fake method to prevent queued jobs from actually being pushed to the queue. After calling the Queue facade's fake method, you may then assert that the application attempted to push jobs to the queue.

Faking a Subset of Jobs

If you only need to fake specific jobs while allowing your other jobs to execute normally, you may pass the class names of the jobs that should be faked to the fake method.

Testing Job Chains

To test job chains, you will need to utilize the Bus facade's faking capabilities.
The Bus facade's assertChained method may be used to assert that a chain of jobs was dispatched.
The assertChained method accepts an array of chained jobs as its first argument:

```
use App\Jobs\RecordShipment;
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Support\Facades\Bus;
Bus::fake();
// ...
Bus::assertChained([
    ShipOrder::class,
    RecordShipment::class,
    UpdateInventory::class
]);
```

As you can see in the example above, the array of chained jobs may be an array of the job's class names. However, you may also provide an array of actual job instances. When doing so, Laravel will ensure that the job instances are of the same class and have the same property values of the chained jobs dispatched by your application:

```
Bus::assertChained([
    new ShipOrder,
    new RecordShipment,
    new UpdateInventory,
]);
```

You may use the assertDispatchedWithoutChain method to assert that a job was pushed without a chain of jobs:

```
Bus::assertDispatchedWithoutChain(ShipOrder::class);
```

Testing Chain Modifications

If a chained job prepends or appends jobs to an existing chain, you may use the job's assertHasChain method to assert that the job has the expected chain of remaining jobs:

```
$job = new ProcessPodcast;
```

```
$job->handle();
$job->assertHasChain([
    new TranscribePodcast,
    new OptimizePodcast,
    new ReleasePodcast,
]);
```

The assertDoesntHaveChain method may be used to assert that the job's remaining chain is empty:

```
$job->assertDoesntHaveChain();
```

Testing Chained Batches

If your job chain contains a batch of jobs, you may assert that the chained batch matches your expectations by inserting a Bus::chainedBatch definition within your chain assertion:

```
use App\Jobs\ShipOrder;
use App\Jobs\UpdateInventory;
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;
Bus::assertChained([
    new ShipOrder,
    Bus::chainedBatch(function (PendingBatch $batch) {
        return $batch->jobs->count() === 3;
    }),
    new UpdateInventory,
]);
```

Testing Job Batches

The Bus facade's assertBatched method may be used to assert that a batch of jobs was dispatched. The closure given to the assertBatched method receives an instance of Illuminate\Bus\PendingBatch, which may be used to inspect the jobs within the batch.

```
use Illuminate\Bus\PendingBatch;
use Illuminate\Support\Facades\Bus;
Bus::fake();
 // ...
Bus::assertBatched(function (PendingBatch $batch) {
    return $batch->name == 'import-csv' &&
           $batch->jobs->count() === 10;
});
```

You may use the assertBatchCount method to assert that a given number of batches were dispatched:

```
Bus::assertBatchCount(3);
```

You may use assertNothingBatched to assert that no batches were dispatched:

```
Bus::assertNothingBatched();
```

Testing Job / Batch Interaction

For example, you may need to test if a job cancelled further processing for its batch. To accomplish this, you need to assign a fake batch to the job via the withFakeBatch method. The withFakeBatch method returns a tuple containing the job instance and the fake batch:

Testing Job / Queue Interactions

Sometimes, you may need to test that a queued job releases itself back onto the queue. Or, you may need to test that the job deleted itself. You may test these queue interactions by instantiating the job and invoking the withFakeQueueInteractions method.

Once the job's queue interactions have been faked, you may invoke the handle method on the job. After invoking the job, the assertReleased, assertDeleted, assertNotDeleted, assertFailed, and assertNotFailed methods may be used to make assertions against the job's queue interactions.

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.

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.