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:**

****

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:

```
public function middleware(): array
{
    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**:

public function middleware(): array

{

return [(new WithoutOverlapping($this->order->id))->dontRelease()];

}

****

The **WithoutOverlapping middleware** is powered by Laravel's atomic lock feature. Sometimes, your job may unexpectedly fail or timeout in such a way that the lock is not released. Therefore, you may explicitly define a lock expiration time using the **expireAfter method**. For example, the example below will instruct Laravel to release the **WithoutOverlapping lock** three minutes after the job has started processing:

```
public function middleware(): array
{
    return [(new WithoutOverlapping($this->order->id))->expireAfter(180)];
}
```

****

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. However, you can instruct Laravel to apply the **key across job classes** using the shared method:

```
class ProviderIsDown
{
    // …
    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}
```

```php
class ProviderIsUp
{
    // ...
    public function middleware(): array
    {
        return [
            (new WithoutOverlapping("status:{$this->provider}"))->shared(),
        ];
    }
}
```

****

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 a 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:

use DateTime;

use Illuminate\Queue\Middleware\ThrottlesExceptions;

public function middleware(): array

{

return [new ThrottlesExceptions(10, 5)];

}

```php
public function retryUntil(): DateTime
{
```

```
    return now()->addMinutes(5);
}
```

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 minutes such a job should be delayed by calling the **backoff method** when attaching the middleware to the job:

public function middleware(): array

{

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":

```
public function middleware(): array
{
    return [(new ThrottlesExceptions(10, 10))->by('key')];
}
```

****

Note: 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.

****

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);
```

In new Laravel applications, the **sync driver** is the default queue driver. This driver executes jobs synchronously in the **foreground** of the current request, which is often convenient during local development. If you would like to actually begin queueing jobs for background processing, you may specify a different queue driver within your application's **config/queue.php** configuration file.

****

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:

```
public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);
        ProcessPodcast::dispatch($podcast)
                ->delay(now()->addMinutes(10));
        return redirect('/podcasts');
    }
```

****

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

****

Alternatively, the **dispatchAfterResponse** method delays dispatching a job until after the HTTP response is sent to the user's browser if your web server is using FastCGI. 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();

// this will execute on the main cmd and print message with serve cmd line

****

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();

****

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:

```
public function store(Request $request): RedirectResponse
    {
        $podcast = Podcast::create(/* ... */);

        // Create podcast...

        ProcessPodcast::dispatchSync($podcast);

        return redirect('/podcasts');
    }
```

****

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**.

****

Note: 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.

****

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:

```
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 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 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();
```

****

Note: 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.

****

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();
```

Note: Here we must set the command in this order, else and exception will raise.

****

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:

Bus::chain([

new ProcessPodcast,

new OptimizePodcast,

new ReleasePodcast,

])->catch(function (Throwable $e) {

// A job within the chain has failed...

})->dispatch();

****

Note (Very Important): 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.

****

you may specify the job's queue by calling the **onQueue method** within the job's constructor:

```
public function __construct()
   {
      $this->onQueue('processing');
   }
```

****

you may specify the job's connection by calling the **onConnection method** within the job's constructor:

```
public function __construct()
   {
      $this->onConnection('sqs');
   }
```

****