## Job Batching

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

***php artisan queue:batches-table***

***php artisan migrate***

**********

## Defining Batchable Jobs

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

```
use Illuminate\Bus\Batchable;

class ImportCsv implements ShouldQueue
{
    use Batchable, Dispatchable, InteractsWithQueue, Queueable, SerializesModels;


    /**
     * 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 Illuminate\Bus\Batch;
$batch = Bus::batch([
    new ImportCsv(1, 100),
    new ImportCsv(101, 200),
    new ImportCsv(201, 300),
    new ImportCsv(301, 400),
    new ImportCsv(401, 500),
])->then(function (Batch $batch) {
    // All jobs completed successfully...
})->catch(function (Batch $batch, Throwable $e) {
    // First batch job failure detected...
})->finally(function (Batch $batch) {
    // The batch has finished executing...
})->dispatch();

return $batch->id;
```

**********

**Note:** 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.

**********

**Note (To Remember):** Since batch callbacks are serialized and executed later by the Laravel queue, you should not use the **$this** variable within the callbacks.

**********

## Naming Batches

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

```
$batch = Bus::batch([

    // ...

])->then(function (Batch $batch) {

    // All jobs completed successfully...

})->name('Import CSV')->dispatch();
```

**********

## Batch Connection & Queue

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

```
$batch = Bus::batch([

])->then(function (Batch $batch) {

})->onConnection('redis')->onQueue('imports')->dispatch();
```

**********

## Chains Within Batches

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

```
Bus::batch([
    [
        new ReleasePodcast(1),
        new SendPodcastReleaseNotification(1),
    ],
    [
        new ReleasePodcast(2),
        new SendPodcastReleaseNotification(2),
    ],
])->then(function (Batch $batch) {
    // ...
})->dispatch();
```

**********

## Adding Jobs To Batches

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

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

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

```
public function handle(): void
{
    if ($this->batch()->cancelled()) {
        return;
    }

    $this->batch()->add(Collection::times(1000, function () {
```

```
        return new ImportContacts;
    }));
}
```

**********

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

**********

## Inspecting Batches

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

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

// The name of the batch (if applicable)...
$batch->name;

// The number of jobs assigned to the batch...
$batch->totalJobs;

// The number of jobs that have not been processed by the queue...
$batch->pendingJobs;
```

**// The number of jobs that have failed...**

**$batch->failedJobs;**

**// The number of jobs that have been processed thus far...**

**$batch->processedJobs();**

**// The completion percentage of the batch (0-100)...**

**$batch->progress();**

**// Indicates if the batch has finished executing...**

**$batch->finished();**

**// Cancel the execution of the batch...**

**$batch->cancel();**

**// Indicates if the batch has been cancelled...**

**$batch->cancelled();**

**********

## Returning Batches From Routes

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

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

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

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

**********

## Cancelling Batches

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

```
public function handle(): void
{
    if ($this->user->exceedsImportLimit()) {
        return $this->batch()->cancel();
    }

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

As you may have noticed in the previous examples, batched jobs should typically determine if their corresponding batch has been cancelled before continuing execution. However, for convenience, you may assign the SkipIfBatchCancelled middleware to the job instead. As its name indicates, this middleware will instruct Laravel to not process the job if its corresponding batch has been cancelled:

```
use Illuminate\Queue\Middleware\SkipIfBatchCancelled;

public function middleware(): array
{
    return [new SkipIfBatchCancelled];
}
```

**********

## Batch Failures

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

**********

## Allowing Failures

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

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

**********

## Retrying Failed Batch Jobs

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

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

**********

## Pruning Batches

Without pruning, the job_batches table can accumulate records very quickly.

To mitigate this, you should schedule the queue:prune-batches Artisan command to run daily:

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

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

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

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:

***$schedule->command('queue:prune-batches --hours=48 --unfinished=72')->daily();***

Likewise, your **jobs_batches** table may also accumulate batch records for cancelled batches. You may instruct the **queue:prune-batches** command to prune these cancelled batch records using the cancelled option:

***$schedule->command('queue:prune-batches --hours=48 --cancelled=72')->daily();***

**********

## Queueing Closures

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

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

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

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

```
use Throwable;

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

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

**********

**Note:** 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.

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

php artisan **queue:listen**

**********

## Running Multiple Queue Workers

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

**********

## Specifying The Connection & Queue

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

***php artisan queue:work redis***

**********

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 (This will execute only one job).***

**********

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

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

**********

## Processing All Queued Jobs & Then Exiting

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

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

**********

## Processing Jobs For A Given Number Of Seconds

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

***# Process jobs for one hour and then exit...***

***php artisan queue:work --max-time=3600***

**********

## Worker Sleep Duration

When jobs are available on the queue, the worker will keep processing jobs with no delay in between 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***

**********

## Resource Considerations

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

**********

## Queue Priorities

Sometimes you may wish to prioritize how your queues are processed. For example, in your config/queue.php configuration file, you may set the default queue for your redis connection to low. However, occasionally you may wish to push a job to a high priority queue like so: ***dispatch((new Job)->onQueue('high'));***

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

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

**********

## Queue Workers & Deployment

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

***php artisan queue:restart***

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

**Note:** 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.

**********