Book Cover

Table of Contents

Queues 101

Every good technical book starts with a crash course, and this one is no exception.

In this part, we will explore the key components of a queue system, see how queues are used in Laravel, and understand why we need to use queues in our applications.

Now we're going to check for that cache key at the beginning of the `handle()` method of our job and release the job back to the queue if the cache key hasn't expired yet:

```php
public function handle()
{
    if ($timestamp = Cache::get('api-limit')) {
        return $this->release(
            $timestamp - time()
        );
    }

    // ...
}
```

`$timestamp - time()` will give us the seconds remaining until requests are allowed.

Here's the whole thing:

```php
public function handle()
{
    if ($timestamp = Cache::get('api-limit')) {
        return $this->release(
            $timestamp - time()
        );
    }

    $response = Http::acceptJson()
        ->timeout(10)
        ->withToken('...')
        ->get('https://...');

    if ($response->failed() && $response->status() == 429) {
        $secondsRemaining = $response->header('Retry-After');

        Cache::put(
            'api-limit',
            now()->addSeconds($secondsRemaining)->timestamp,
            $secondsRemaining
        );

        return $this->release(
            $secondsRemaining
        );
    }

    // ...
}
```

Notice: In this part of the challenge we're only handling the 429 request error. In the actual implementation, you'll need to handle other 4xx and 5xx errors as well.

Replacing the Tries Limit with Expiration

Since the request may be throttled multiple times, it's better to use the job expiration configuration instead of setting a static tries limit.

```php
public $tries = 0;

// ...

public function retryUntil()
{
    return now()->addHours(12);
}
```

Now if the job was throttled by the limiter multiple times, it will not fail until the 12-hour period passes.

Limiting Exceptions

In case an unhandled exception was thrown from inside the job, we don't want it to keep retrying for 12 hours. For that reason, we're going to set a limit for the maximum exceptions allowed:

```php
public $tries = 0;
public $maxExceptions = 3;
```

Now the job will be attempted for 12 hours, but will fail immediately if 3 attempts failed due to an exception or a timeout.

Handling Queues on Deployments

> **Notice:** This is a sample content from Laravel Queues in Action. A book by Mohamed Said the creator of Ibis.

When you deploy your application with new code or different configurations, workers running on your servers need to be informed about the changes. Since workers are long-living processes, they must be shut down and restarted in order for the changes to be reflected.

Restarting Workers Through The CLI

When writing your deployment script, you need to run the following command after pulling the new changes:

```
php artisan queue:restart
```

This command will send a signal to all running workers instructing them to exit after finishing any job in hand. This is called "graceful termination".

If you're using Laravel Forge, here's a typical deployment script that you may use:

```
cd /home/forge/mysite.com
git pull origin master
$FORGE_COMPOSER install --no-interaction --prefer-dist --optimize-
autoloader

( flock -w 10 9 || exit 1
    echo 'Restarting FPM...'; sudo -S service $FORGE_PHP_FPM reload )
9>/tmp/fpmlock

$FORGE_PHP artisan migrate --force
$FORGE_PHP artisan queue:restart
```

Here the new code will be pulled from git, dependencies will be installed by composer, php-fpm will be restarted, migrations will run, and finally, the queue restart signal will be sent.