Book Cover

Table of Contents

Queues 101

this is the second queues 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.

Key Components

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

A queue system has 3 main components; queues, messages, and workers.

Let's start by exploring each of these components.

Queues (سلام)

> *A queue is a linear data structure in which elements can be added to one end, and can only be removed from the other end.*

That's the definition you find in most Computer Science books, and it can be quite confusing.

Let me show you a queue in action:

```
$queue = [
    'DownloadProject',
    'RunTests'
];
```

This queue contains two messages. We can enqueue a new message and it'll be added to the end of the queue:

```
enqueue('Deploy');

$queue = [
    'DownloadProject',
    'RunTests',
    'Deploy'
];
```

This is the function for `eating`

```
class Person{
    public function eat(){
        return 'they eat every day '
    }
}
```

We can also `dequeue` a message and it'll be removed from the beginning of the queue:

```
$message = dequeue(); // == DownloadProject

$queue = [
    'RunTests',
    'Deploy'
];
```

If you ever heard the term "first-in-first-out (FIFO)" and didn't understand what it means, now you know it means the first message in the queue is the first message that gets processed.

Messages

A message is a call-to-action trigger. You send a message from one part of your application and it triggers an action in another part. Or you send it from one application and it triggers an action in a completely different application.

The message sender doesn't need to worry about what happens after the message is sent, and the message receiver doesn't need to know who the sender is.

A message body contains a string, this string is interpreted by the receiver and the call to action is extracted.

Workers

A worker is a long-running process that dequeues messages and executes the call-to-action.

Here's an example worker:

```
while (true) {
    $message = dequeue();

    processMessage(
        $message
    );
}
```

Once you start a worker, it'll keep dequeuing messages from your queue. You can start as many workers as you want; the more workers you have, the faster your messages get dequeued.

Queues in Laravel

Laravel ships with a powerful queue system right out of the box. It supports multiple drivers for storing messages:

- Database
- Beanstalkd
- Redis
- Amazon SQS

Enqueuing messages in Laravel can be done in several ways, and the most basic method is using the `Queue` facade:

```
use Illuminate\Support\Facades\Queue;

Queue::pushRaw('Send invoice #1');
```

If you're using the database queue driver, calling `pushRaw()` will add a new row in the `jobs` table with the message "Send invoice #1" stored in the `payload` field.

Enqueuing raw string messages like this is not very useful. Workers won't be able to identify the action that needs to be triggered. For that reason, Laravel allows you to enqueue class instances:

```
use Illuminate\Support\Facades\Queue;

Queue::push(
    new SendInvoice(1)
);
```

> **Notice:** Laravel uses the term "push" instead of "enqueue", and "pop" instead of "dequeue".

When you enqueue an object, the queue manager will serialize it and build a string payload for the message body. When workers dequeue that message, they will be able to extract the object and call the proper method to trigger the action.

Laravel refers to a message that contains a class instance as a "Job". To create a new job in

your application, you may run this artisan command:

```
php artisan make:job SendInvoice
```

This command will create a `SendInvoice` job inside the `app/Jobs` directory. That job will look like this:

```
namespace App\Jobs;

class SendInvoice implements ShouldQueue
{
    use Dispatchable;
    use InteractsWithQueue;
    use Queueable;
    use SerializesModels;

    public function __construct()
    {
    }

    public function handle()
    {
        // Execute the job logic.
    }
}
```

When a worker dequeues this job, it will execute the `handle()` method. Inside that method, you should put all your business logic.

> **Notice:** Starting Laravel 8.0, you can use `__invoke` instead of `handle` for the method name.

The Command Bus

Laravel ships with a command bus that can be used to dispatch jobs to the queue. Dispatching through the command bus allows us to use several functionalities that I'm going to show you later.

Throughout this book, we're going to use the command bus to dispatch our jobs instead of

the `Queue::push()` method.

Here's an example of using the `dispatch()` helper which uses the command bus under the hood:

```
dispatch(
    new SendInvoice(1)
);
```

Or you can use the `Bus` facade:

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

Bus::dispatch(
    new SendInvoice(1)
);
```

You can also use the `dispatch()` static method on the job class:

```
SendInvoice::dispatch(1);
```

> **Notice:** Arguments passed to the static `dispatch()` method will be transferred to the job instance automatically.

Starting A Worker

To start a worker, you need to run the following artisan command:

```
php artisan queue:work
```

This command will bootstrap an instance of your application and keep checking the queue for jobs to process.

```
$app = require_once __DIR__.'/bootstrap/app.php';

while (true) {
    $job = $app->dequeue();

    $app->process($job);
}
```

Re-using the same instance of your application has a major performance gain as your server will have to bootstrap the application only once during the time the worker process is alive. We'll talk more about that later.

Why Use a Message Queue?

Remember that queuing is a technique for indirect program-to-program communication. Your application can send messages to the queue, and other parts of the application can read those messages and execute them.

Here's how this can be useful:

Better User Experience

The most common reason for using a message queue is that you don't want your users to wait for certain actions to be performed before they can continue using your application.

For example, a user doesn't have to wait until your server communicates with a third-party mail service before they can complete a purchase. You can just send a success response so the user continues using the application while your server works on sending the invoice in the background.

Fault Tolerance

Queue systems are built to persist jobs until they are processed. In the case of failure, you can configure your jobs to be retried several times. If the job keeps failing, the queue manager will put it in safe storage so you can manually intervene.

For example, if your job interacts with an external service and it went down temporarily, you can configure the job to retry after some time until this service is back online.

Scalability

For applications with unpredictable load, it's a waste of money to allocate more resources all the time even when there's no load. With queues, you can control the rate at which your workers consume jobs.

Allocate more resources and your jobs will be consumed faster, limit those resources and your jobs will be consumed slower but you know they'll eventually be processed.

Batching

Batching a large task into several smaller tasks is more efficient. You can tune your queues to process these smaller tasks in parallel which will guarantee faster processing.

Ordering and Rate Limiting

With queues, you can ensure that certain jobs will be processed in a specific order. You can also limit the number of jobs running concurrently.

Cookbook

In this part, we will walk through several real-world challenges with solutions that actually run in production. These are challenges I met while building products at Laravel and others collected while supporting the framework users for the past four years.

While doing this, I'm going to explain every queue configuration, gotcha, and technique we meet on our way.

Canceling Abandoned Orders

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

When users add items to their shopping cart and start the checkout process, you want to reserve these items for them. However, if a user abandoned an order—they never canceled or checked out—you will want to release the reserved items back into stock so other people can order them.

To do this, we're going to schedule a job once a user starts the checkout process. This job will check the order status after **an hour** and cancel it automatically if it wasn't completed by then.

Delay Processing a Job

Let's see how such a job can be dispatched from the controller action:

```
class CheckoutController
{
    public function store()
    {
        $order = Order::create([
            'status' => Order::PENDING,
            // ...
        ]);

        MonitorPendingOrder::dispatch($order)->delay(3600);
    }
}
```

By chaining the `delay(3600)` method after `dispatch()`, the `MonitorPendingOrder` job will be pushed to the queue with a delay of 3600 seconds (1 hour); workers will not process this job before the hour passes.

You can also set the delay using a `DateTimeInterface` implementation:

```
MonitorPendingOrder::dispatch($order)->delay(
    now()->addHour()
);
```

> **Warning:** Using the SQS driver, you can only delay a job for 15 minutes. If you want to delay jobs for more, you'll need to delay for 15 minutes first and then keep releasing the job back to the queue using `release()`. You should also know that SQS stores the job for only 12 hours after it was enqueued.

Here's a quick look inside the `handle()` method of that job:

```
public function handle()
{
    if ($this->order->status == Order::CONFIRMED ||
        $this->order->status == Order::CANCELED) {
        return;
    }

    $this->order->markAsCanceled();
}
```

When the job runs—after an hour—, we'll check if the order was canceled or confirmed and just return from the `handle()` method. Using `return` will make the worker consider the job as successful and remove it from the queue.

Finally, we're going to cancel the order if it was still pending.

Sending Users a Reminder Before Canceling

It might be a good idea to send the user an SMS notification to remind them about their order before completely canceling it. So let's send an SMS every 15 minutes until the user completes the checkout or we cancel the order after 1 hour.

To do this, we're going to delay dispatching the job for 15 minutes instead of an hour:

```
MonitorPendingOrder::dispatch($order)->delay(
    now()->addMinutes(15)
);
```

When the job runs, we want to check if an hour has passed and cancel the order.

If we're still within the hour period, then we'll send an SMS reminder and release the job back to the queue with a 15-minute delay.

```
public function handle()
{
    if ($this->order->status == Order::CONFIRMED ||
        $this->order->status == Order::CANCELED) {
        return;
    }

    if ($this->order->olderThan(59, 'minutes')) {
        $this->order->markAsCanceled();

        return;
    }

    SMS::send(...);

    $this->release(
        now()->addMinutes(15)
    );
}
```

Using `release()` inside a job has the same effect as using `delay()` while dispatching. The job will be released back to the queue and workers will run it again after 15 minutes.

Ensuring the Job Has Enough Attempts

Every time the job is released back to the queue, it'll count as an attempt. We need to make sure our job has enough `$tries` to run 4 times:

```
class MonitorPendingOrder implements ShouldQueue
{
    public $tries = 4;
}
```

This job will now run:

```
15 minutes after checkout
30 minutes after checkout
45 minutes after checkout
60 minutes after checkout
```

If the user confirmed or canceled the order say after 20 minutes, the job will be deleted from the queue when it runs on the attempt at 30 minutes and no SMS will be sent.

This is because we have this check at the beginning of the `handle()` method:

```
if ($this->order->status == Order::CONFIRMED ||
    $this->order->status == Order::CANCELED) {
    return;
}
```

A Note on Job Delays

There's no guarantee workers will pick the job up exactly after the delay period passes. If the queue is busy and not enough workers are running, our `MonitorPendingOrder` job may not run enough times to send the 3 SMS reminders before canceling the order.

To increase the chance of your delayed jobs getting processed on time, you need to make sure you have enough workers to empty the queue as fast as possible. This way, by the time the job becomes available, a worker process will be available to run it.

Dealing With API Rate Limits

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

If your application communicates with 3rd party APIs, there's a big chance some rate limiting strategies are applied. Let's see how we may deal with a job that sends an HTTP request to an API that only allows 30 requests per minute:

Here's how the job may look:

```
public $tries = 10;

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

    if ($response->failed() && $response->status() == 429) {
        return $this->release(30);
    }

    // ...
}
```

If we hit a rate limit response `429 Too Many Requests`, we're going to release the job back to the queue to be retried again after 30 seconds. We also configured the job to be retried 10 times.

Not Sending Requests That We Know Will Fail

When we hit the limit, any requests sent before the limit reset point will fail. For example, if we sent all 30 requests at 10:10:45, we won't be able to send requests again before 10:11:00.

If we know requests will keep failing, there's no point in sending them and delaying processing other jobs in the queue. Instead, we're going to set a key in the cache when we

hit the limit, and release the job right away if the key hasn't expired yet.

Typically when an API responds with a 429 response code, a `Retry-After` header is sent with the number of seconds to wait before we can send requests again:

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

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

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

Here we set an `api-limit` cache key with an expiration based on the value from the `Retry-After` header.

The value stored in the cache key will be the timestamp when requests are going to be allowed again:

```
now()->addSeconds($secondsRemaining)->timestamp
```

We're also going to use the value from `Retry-After` as a delay when releasing job:

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

That way the job is going to be available as soon as requests are allowed again.

> **Warning:** When dealing with input from external services—including headers—it might be a good idea to validate that input before using it.

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:

```
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:

```
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.

```
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:

```
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.

After `php-fpm` is restarted, your application visitors will start using the new code while the workers are still running on older code. Eventually, those workers will exit and be started again by Supervisor. The new worker processes will be running the new code.

If you're using Envoyer, then you need to add a deployment hook after the “Activate New Release” action and run the `queue:restart` command.

Restarting Workers Through Supervisor

If you have the worker processes managed by Supervisor, you can use the `supervisorctl` command-line tool to restart them:

```
supervisorctl restart group-name:*
```

> **Notice:** A more detailed guide on configuring Supervisor is included.

Restarting Horizon

Similar to restarting regular worker processes, you can signal Horizon's master supervisor to terminate all worker processes by using the following command:

```
php artisan horizon:terminate
```

But in order to ensure your jobs won't be interrupted, you need to make sure of the following:

1. Your Horizon supervisors' `timeout` value is greater than the number of seconds consumed by the longest-running job.
2. Your job-specific `timeout` is shorter than the timeout value of the Horizon supervisor.
3. If you're using the Supervisor process manager to monitor the Horizon process, make sure the value of `stopwaitsecs` is greater than the number of seconds consumed by the longest-running job.

With this correctly configured, Supervisor will wait for the Horizon process to terminate and won't force-terminate it after `stopwaitsecs` passes.

Horizon supervisors will also wait for the longest job to finish running and won't force-terminate after the timeout value passes.

Dealing With Migrations

When you send a restart signal to the workers, some of them may not restart right away; they'll wait for a job in hand to be processed before exiting.

If you are deploying new code along with migrations that'll change the database schema, workers that are still using the old code may fail in the middle of running their last job due to those changes; old code working with the new database schema!

To prevent this from happening, you'll need to signal the workers to exit and then wait for them. Only when all workers exit gracefully you can start your deployment.

To signal the workers to exit, you'll need to use `supervisorctl stop` in your deployment script. This command will block the execution of the script until all workers are shutdown:

```
sudo supervisorctl stop group-name:*

cd /home/forge/mysite.com
# ...

$FORGE_PHP artisan migrate --force

sudo supervisorctl start group-name:*
```

> **Warning:** Make sure the system user running the deployment can run the `supervisorctl` command as `sudo`.

Now, your workers will be signaled by Supervisor to stop after processing any jobs in hand. After all workers exit, the deployment script will continue as normal; migrations will run, and finally, the workers will be started again.

However, you should know that `supervisorctl stop` may take time to execute depending on how many workers you have and if any long-running job is being processed.

You don't want to stop the workers in this way if you don't have migrations that change the schema. So, I recommend that you don't include `supervisorctl stop` in your deployment script by default. Only include it when you know you're deploying a migration that will change the schema and cause workers running on old code to start throwing exceptions.

You can also manually run `supervisorctl stop`, wait for the command to execute, start

the deployment, and finally run `supervisorctl start` after your code deploys.