ScopeTreasure Courses

LARAVEL COURSES · LESSON LV.16 · CODE SCOPE SERIES

Queues & Jobs — Background Mein Kaam

Slow work moves backstage: job classes, dispatch() vs the worker, the database queue driver, failed_jobs and retries, delayed jobs.

Introduction

LV.15 ended with a cost: sending the welcome email can hold a user's page hostage for 2–3 seconds while the mail server thinks. Multiply that by report cards for a whole class, and pages time out. The user does not care about your emails — they care that *registration finished*. So move the slow work **backstage**.

A **queue** is a waiting line of tasks. A **job** is one task written as a class. Your page does not DO the task — it writes a note ("send Fatima's welcome email") onto the line and answers the user instantly. A separate process, the **queue worker**, reads notes one by one and does the work where nobody is kept waiting. This lesson moves the welcome letter backstage, watches the worker perform it, and handles the day a job fails.

(Roman Urdu: Safay ko mehman ke saath rakho; dheema kaam pichlay kamray mein — note likh do, kaam khud ho jaye ga.)

Key points

- **Dispatch is not doing:** `SendWelcomeEmail::dispatch($user)` writes the job into the queue and returns *immediately*. The page feels instant — because the work has not happened yet. It happens when the worker gets there. Confusing these two moments is the root of every queue misunderstanding.
- **A job is a class with one `handle()`:** `php artisan make:job SendWelcomeEmail` — the constructor receives what the job needs (the user), `handle()` does the slow thing (sends LV.15's mail). Jobs implement `ShouldQueue` — that marker is what tells Laravel "this belongs on the line, not in the page".
- **The driver is where the line lives:** `.env`'s `QUEUE_CONNECTION` — `sync` (no line at all; do it now, the default in a fresh app), `database` (notes stored as rows in a `jobs` table — simple, honest, perfect to learn on), `redis` (the production favourite). Modern Laravel's default tables migration already includes the `jobs` and `failed_jobs` tables.
- **The worker is a living process:** `php artisan queue:work` starts a clerk who never sleeps: reads a note, runs `handle()`, marks it done, reads the next. If the worker is not running, NOTHING moves — the line grows quietly while pages smile. On a server, the worker is kept alive by a supervisor; on your laptop, by a terminal you leave open.

  (Roman Urdu: Line apne aap nahi chalti — peechay aik bunda khara rakho jo note parh kar kaam karta rahe. Wo so gaya to line ruk gayi.)
- **Failures are recorded, not lost:** a job that throws (mail server down) lands in `failed_jobs` with its error written down. Fix the cause, then `php artisan queue:retry all` puts the notes back on the line. Nothing silently evaporates.

- **Jobs can be told "later":** `->delay(now()->addMinutes(10))` parks a note for ten minutes — the "we noticed you left items in your cart" letter, the report generated at midnight. The line holds time as politely as it holds work.
- **The page-vs-queue split is a design question:** ask of every task — *must the user SEE its result to continue?* Payment confirmation: page. Welcome email, thumbnails, exports, notifications: queue. Most slow things belong backstage.

Code examples

Example 1 — The job class: LV.15's letter, wrapped for backstage

```
php artisan make:job SendWelcomeEmail
```

```
// app/Jobs/SendWelcomeEmail.php
class SendWelcomeEmail implements ShouldQueue
{
    use Queueable;

    public function __construct(public User $user) {}

    public function handle(): void
    {
        Mail::to($this->user->email)->send(new WelcomeMail($this->user));
    }
}
```

```
// registration — the page now writes a note and walks away
SendWelcomeEmail::dispatch($user);
```

What it does: The mailable did not change AT ALL — queues wrap work, they do not rewrite it. The job's whole personality: receive the user, and in `handle()`, send the letter. Registration returns the instant the note is dispatched.

What you will see: with the worker OFF, register a teacher and time the page: it returns instantly — and no letter arrives in Mailpit yet. The note is written; the work waits. Example 2 wakes the clerk.

Example 2 — The database line and the worker's shift

```
# .env
QUEUE_CONNECTION=database
```

What it does: With the database driver, every dispatched job is a row in the `jobs` table — open phpMyAdmin after a registration and read the note itself: its payload names the job class and the user. Then start the

worker in its own terminal and register one more teacher.

What you will see: the worker's terminal wakes the moment the note lands:

```
2026-01-19 10:02:41 App\Jobs\SendWelcomeEmail ..... RUNNING
2026-01-19 10:02:44 App\Jobs\SendWelcomeEmail ..... DONE (3.01s)
```

Three honest seconds of mail-server time — spent backstage, while the user's page had already finished. The row leaves the `jobs` table, and Fatima's letter sits in Mailpit. Keep that terminal open: it is a staff member on shift, not a command that "ran".

Example 3 — The bad day: failure, the record, and the retry

```
php artisan queue:failed        # who is waiting in the failure ward?
php artisan queue:retry all     # healed — back onto the line
```

What it does: Stop Mailpit (simulate the mail server dying) and register a teacher. The job's `handle()` throws; the queue does not lose the note — it moves it to `failed_jobs` with the exception written beside it. Start Mailpit again (the cause is healed) and retry.

What you will see: `queue:failed` lists the casualty — job name, connection, and the connection-refused error in plain text. After `queue:retry all`, the worker picks the note up, `DONE` prints, and the letter arrives late but alive. Nothing about the user or the data was lost — the line simply waited for a healthier world.

Quick recap

- A job is a `ShouldQueue` class: constructor takes the ingredients, `handle()` does the slow work.
- `dispatch()` writes the note instantly; the **worker** (`queue:work`) performs it — no worker, no movement.
- The database driver stores the line in `jobs`; failures rest in `failed_jobs` until `queue:retry`.
- Delays park jobs in time; ask of every task: page, or backstage?

Common mistakes

1. **The worker is not running** — the number-one queue bug report: "dispatch works, nothing happens." Fix: check the clerk first — `queue:work` must be alive (on servers, under a supervisor that restarts it).
2. **Believing dispatch means done** — code after `dispatch()` that assumes the email was sent, the file processed, the user notified. Fix: dispatch means *promised*; design as if the work completes seconds or minutes later, because it can.
3. **Passing giant or dying data into jobs** — stuffing whole uploaded files or stale model copies into the constructor. Fix: pass models (Laravel serialises them to an id and re-fetches fresh) or small values; the job reads the world as it is at run time.
4. **Ignoring `failed_jobs` forever** — failures accumulate silently until the missing letters become a complaint. Fix: read the failure ward regularly (and later, alert on it); a failed job is a letter unsent, not a problem deleted.

Practice

1. Time both worlds: with `QUEUE_CONNECTION=sync`, register and measure the page; switch to `database` + worker and measure again. Write the two numbers on the job class as a comment — future-you deserves the evidence.
2. Make a `SendMarksUpdatedJob` for LV.15's marks letter with a `->delay(now()->addMinutes(5))`. Change a student's marks, watch the note sit in the `jobs` table, and catch the letter arriving in Mailpit five minutes later — on time, by itself.

Practice quiz

1. What is the difference between dispatch and doing in queues?

- There is none
- SendWelcomeEmail::dispatch($user) writes the job into the queue and returns immediately; the work happens later, when the worker gets there
- Dispatch does the work, the worker only logs it
- Dispatch is for mail only

Answer: SendWelcomeEmail::dispatch($user) writes the job into the queue and returns immediately; the work happens later, when the worker gets there

2. If the worker is not running, what happens to dispatched jobs?

- They run by themselves
- Nothing moves: the line grows quietly while pages smile, because queue:work is the living process that reads a note, runs handle(), marks it done, and reads the next
- They run on the next page load
- They are deleted

Answer: Nothing moves: the line grows quietly while pages smile, because queue:work is the living process that reads a note, runs handle(), marks it done, and reads the next

3. A job throws because the mail server is down. Where does it go, and how do you retry it?

- It vanishes silently
- It lands in failed_jobs with its error written down; fix the cause, then php artisan queue:retry all puts the notes back on the line
- It retries forever by itself
- It blocks the queue forever

Answer: It lands in failed_jobs with its error written down; fix the cause, then php artisan queue:retry all puts the notes back on the line

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/laravel-queues-jobs · More free tools and notes at https://scopetreasure.com