Middleware Deep — Har Darwazay Ka Guard

Guards on the road: handle() and $next, aliases in bootstrap/app.php, three ways to attach, an admin guard with 403, parameters and the order rule.

Introduction

You have already met a guard: LV.7's `auth` middleware stood at the Students door checking "logged in?" before any controller woke. **Middleware** is the general idea — a checkpoint a request must pass on its way in (and sometimes on its way out). Age checks, admin roles, language setters, loggers: each is a small class with one question to ask, and Laravel lines them up in order around your routes.

This lesson builds guards of your own: the anatomy (`handle()` and the all-important `$next`), naming them in modern Laravel's `bootstrap/app.php`, the three ways to attach a guard to a road, a working **admin-only** checkpoint that answers 403, guards that take parameters, and the order rule that decides which guard asks first.

(Roman Urdu: Controller daftar hai; middleware raste ke chowkidaar hain — daftar tak wahi pahunche ga jo har chowki se guzar jaye.)

Key points

- **Anatomy is two moves:** `handle($request, Closure $next)` either stops the request (return a response yourself — a redirect, a 403) or passes it onward with `return $next($request);` — "proceed to the next checkpoint or the controller". Every guard you ever write is these two moves in costume.
- **Making one is one command:** `php artisan make:middleware CheckAge` creates the class in `app/Http/Middleware/`. Registration in modern Laravel happens in **`bootstrap/app.php`**, inside `->withMiddleware(...)`, where you give the class a short **alias** — the nickname routes will use.
- **Three ways to attach a guard:** on ONE route (`->middleware('age')`), on a GROUP (LV.7's way — the whole mohalla behind one checkpoint), or in a CONTROLLER's constructor (the guard travels with the controller wherever it is routed). Same guard, three scopes.
- **A guard answers with status codes:** blocking is returning a response — `abort(403)` for "we know you, and no", redirects for "not yet". 403 is the guard's honest word: Forbidden.

  (Roman Urdu: Guard ghussa nahi karta, jhooth nahi bolta — jo rukne ka haqdar hai, use ruk kar wajah de deta hai: 403.)
- **Parameters make one guard many:** `middleware('role:admin')` hands the guard a value after the colon; one `role` guard then serves admin-only, teacher-only and any future role, instead of three near-identical classes.
- **`terminate()` runs after the answer left:** logging response times, recording visits — work that must not slow the user happens here, once the response is already travelling. The guard clocks out after the guest has gone.

- **Order is logic:** guards run in the order attached, outside-in. Identity before permission: `auth` must establish WHO before `role` asks WHAT THEY MAY DO — a role guard meeting a guest has nobody to judge and can only misjudge.

Code examples

Example 1 — The anatomy: CheckAge, and its alias

```
php artisan make:middleware CheckAge
```

```
// app/Http/Middleware/CheckAge.php
class CheckAge
{
    public function handle(Request $request, Closure $next, int $minimum = 18): Response
    {
        if ($request->input('age') < $minimum) {
            return response('This course needs age ' . $minimum . '+.', 403);
        }

        return $next($request);
    }
}
```

```
// bootstrap/app.php — the guard gets its nickname
->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'age'  => \App\Http\Middleware\CheckAge::class,
        'role' => \App\Http\Middleware\EnsureRole::class,
    ]);
})
```

What it does: Read `handle()` like a chowkidaar's two hands: one hand raised (return a 403 response — the request dies here, the controller never hears of it), one hand waving through (`$next`). The third parameter is the guard's appetite for parameters — `middleware('age:21')` would feed it 21 instead of the default 18. `bootstrap/app.php` is simply where aliases are enrolled; routes speak nicknames, never class names.

What you will see: attach it to a course-entry route and submit the form with age 15 — the controller is never reached; the visitor gets the plain 403 sentence. Age 19 passes the raised hand and the form proceeds as if no guard existed.

Example 2 — The admin guard: protecting the report room

```
// app/Http/Middleware/EnsureRole.php
class EnsureRole
{
    public function handle(Request $request, Closure $next, string $role): Response
    {
        if ($request->user()?->role !== $role) {
            abort(403);
        }

        return $next($request);
    }
}
```

```
// routes/web.php — order speaks: identity first, permission second
Route::middleware(['auth', 'role:admin'])->group(function () {
    Route::get('/admin/class-report', [ReportController::class, 'index']);
});
```

What it does: The users table carries a `role` column (`teacher` / `admin`) — LV.7's login proves identity, this guard judges it. The null-safe `?->` matters: a guest's `user()` is null, and the guard must answer 403, not crash. `auth` stands FIRST in the list, so by the time `role` asks its question, there is always a someone to judge (guests were already redirected to login by `auth`).

What you will see: log in as a plain teacher and visit `/admin/class-report` — the honest wall: **403 | This action is unauthorized.** Log in as the admin — the report opens. Same address, two fates, decided before the controller stirred.

Example 3 — terminate(): the guard who writes the diary

```
class CheckAge
{
    public function handle(Request $request, Closure $next): Response
    {
        return $next($request);   // let them through first...
    }

    public function terminate(Request $request, Response $response): void
    {
        // ...then, with the answer already gone, write the diary
        Log::info('Course entry visited', [
            'path'   => $request->path(),
            'status' => $response->getStatusCode(),
        ]);
    }
}
```

What it does: `terminate()` is Laravel's after-shift: it runs when the response has been sent — perfect for measurements and diaries that must never tax the visitor's wait. Pair it with LV.8's log-reading habit and every guard becomes also a witness.

What you will see: visit the course-entry page twice, then open `storage/logs/laravel.log` — two fresh lines, path and status recorded, written after each visitor had already received their page. The diary grows; nobody's page slowed by a single millisecond for it.

Quick recap

- Middleware = checkpoint classes: stop the request with a response, or wave it on with `$next($request)`.
- Aliases are enrolled in `bootstrap/app.php`; attach guards per-route, per-group, or in the controller — identity before permission.
- Parameters (`role:admin`) make one guard serve many doors; `terminate()` does diary work after the response leaves.

Common mistakes

1. **Forgetting `$next` on the happy path** — a guard that checks and then returns nothing turns away EVERYONE, including the allowed. Fix: every path through `handle()` ends in exactly one of two sentences: a blocking response, or `return $next($request);`.
2. **Registering the class but never aliasing (or vice versa)** — routes calling an unknown nickname error loudly at boot, which is good; silently unattached guards are the dangerous kind. Fix: after wiring, prove the guard with a forbidden visit (Example 2's 403) — a guard untested is a rumour.

3. **Permission before identity** — `role` running before `auth` meets guests as null and either crashes or, worse, waves them through a sloppy check. Fix: the order rule is law — authentication first, authorisation second, always.
4. **Doing controller work inside a guard** — fetching reports, formatting pages, sending mail from `handle()`. Fix: guards ask ONE question and step aside; judging is their job, working is the controller's.

Practice

1. Build a `SetLanguage` middleware that reads the LV.17 `lang` cookie and sets the app locale for the request. Attach it to the web group and show the dashboard greeting change with the cookie — no other code touched.
2. Extend EnsureRole usage: protect the destroy route alone with `role:admin` at route level (teachers may add and edit, only admin deletes). Prove it three ways: teacher delete attempt (403), admin delete (flash success), guest delete attempt (login redirect).

Practice quiz

1. What are the only two moves a middleware handle() can end with?

- Return a view, or return a redirect
- Stop the request with a response of its own (a redirect, a 403), or pass it onward with return $next($request); every path ends in exactly one of these two
- Log the request, or ignore it
- Throw an error, or sleep

Answer: Stop the request with a response of its own (a redirect, a 403), or pass it onward with return $next($request); every path ends in exactly one of these two

2. In what order must auth and role guards run, and why?

- Role first, then auth, because permission is more important
- Identity before permission: auth must establish WHO before role asks WHAT THEY MAY DO, or a role guard meeting a guest has nobody to judge
- Order makes no difference
- They run at the same time

Answer: Identity before permission: auth must establish WHO before role asks WHAT THEY MAY DO, or a role guard meeting a guest has nobody to judge

3. What does middleware('role:admin') hand the guard?

- A new guard class
- A parameter after the colon, so one role guard serves admin-only, teacher-only and any future role, instead of three near-identical classes
- The admin user itself

- A database row

Answer: A parameter after the colon, so one role guard serves admin-only, teacher-only and any future role, instead of three near-identical classes

---

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-middleware · More free tools and notes at https://scopetreasure.com