GUL'S NOTEBOOK · ENGINEERING · OPEN ACCESS

My code review checklist for junior Laravel developers

Muhammad Gulzaib
https://orcid.org/0009-0000-8709-6989
SynergyFlow Labs · Islamabad, Pakistan

Published	2026-05-18
URL	https://gulzaib.vercel.app/blog/my-code-review-checklist-for-junior-laravel-developers
Licence	Creative Commons Attribution 4.0 International (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

ABSTRACT

A Laravel code review checklist covering authorisation, validation, queries, migrations, queues and constructive feedback, with before-and-after code.

Keywords: laravel, php, testing, security, engineering

> **TL;DR**
>
> Review the expected behaviour first, then check access control, tenant boundaries and writes before spending time on style. Follow with queries, migrations, jobs and tests. Explain the risk in review comments and distinguish required fixes from optional suggestions.

This guide was prepared in November 2026. The displayed 18 May 2026 date was requested for the post; it is not its original publication date.

This is a proposed checklist and a fictional review example, not a record of a particular team's reviews or measured mentoring results. It is useful for junior and experienced developers alike; the difference is how much context and support the reviewer offers.

Start with the behaviour

Before opening individual files, read the pull request description. What should the user be able to do, and what should stay unchanged? If the acceptance criteria are unclear, ask for an example rather than inferring the requirements from the implementation.

For a school fee system, a change to an invoice description should answer more than "does the form save?" Can an authorised user update the correct invoice? What happens when the invoice belongs to another school or the input is invalid? Those questions shape both the review and the tests.

Check the risky parts first

I would review in this order:

1. Access and tenant boundaries.
2. Validation and writes.
3. Queries and data changes.
4. Jobs and failure handling.
5. Tests and review feedback.

Download the copyable Laravel code review checklist. Mark items that do not apply rather than forcing every pull request into the same template.

Access and tenant boundaries

Authentication establishes who is making the request; authorisation decides what that user may do. Laravel policies and gates provide a place for action-specific decisions. [1] Check the actual server-side route and policy, not only whether a button is hidden in the interface.

For a multi-tenant application, follow every identifier from the request to the query. An invoice lookup, its related student, a download and a background job all need the appropriate school boundary. Check what happens when an authenticated user supplies another tenant's ID. Tenant membership alone may not grant permission to update: the policy must also check the relevant role or permission.

Validation and writes

Check required fields, allowed values, lengths and ownership of related IDs. Validation helps reject invalid input; it does not itself establish permission to act. Laravel provides request validation and access to validated data. [2]

Write only fields the endpoint intends to change. Check server-rendered output escaping, file upload rules and any raw SQL. Confirm that secrets or private records are not exposed in logs, error messages or responses. A model's mass-assignment settings are a useful defence, but should not be the only decision about what the endpoint writes.

A before-and-after review example

The following snippets illustrate a fictional invoice update. They are not complete applications or copies from a real code review. Assume an authenticated route, a trustworthy single `school_id` on the user, an `Invoice` model with `description` fillable, and an `InvoicePolicy` registered for the `update` action. The policy must check the actual business permission, including school membership. Adapt those assumptions to your own tenancy model.

Before review:

`InvoiceController.php (before)`

```
public function update(Request $request, int $id)
{
    $invoice = Invoice::findOrFail($id);
    $invoice->update($request->all());

    return response()->json($invoice);
}
```

The lookup has no explicit school boundary, there is no action authorisation, and all request fields are offered for mass assignment. Existing global scopes, middleware or model guards might mitigate some risks in a real application, so inspect those too rather than assuming this snippet is the whole system.

A possible revision:

`InvoiceController.php (after)`

```
public function update(Request $request, int $id)
{
    $invoice = Invoice::query()
        ->where('school_id', $request->user()->school_id)
        ->findOrFail($id);

    Gate::authorize('update', $invoice);

    $data = $request->validate([
        'description' => ['required', 'string', 'max:500'],
    ]);

    $invoice->update(['description' => $data['description']]);

    return response()->json([
        'id' => $invoice->id,
        'description' => $invoice->description,
    ]);
}
```

The snippet assumes imports for `Request`, `Invoice` and the `Gate` facade. An out-of-school ID returns 404 from the scoped lookup; an invoice in the school can still produce 403 when the policy denies the action. The route must require authentication, and a missing or incorrectly permissive policy would still be a problem. This is illustrative code; it has not been run against a Laravel application for this guide.

Review tests for at least: a permitted update; a same-school user without permission; an invoice in another school; invalid input; and an attempt to submit fields such as `school_id`.

Assert the stored record remains unchanged on rejected requests and that unapproved fields cannot be changed on an otherwise successful request. Laravel's database testing tools support factories and database assertions, though the test setup must match the application. [5]

Queries, migrations and failure paths

A list page may work with five records and slow down with thousands. Look for relationship access inside loops that causes extra queries, unbounded `get()` calls and filters that need suitable indexes. Laravel documents eager loading for relationships; use it where the response needs related records and verify the query behaviour rather than adding it everywhere. [3]

Check migrations against existing data, not only an empty local database. Is a new required column compatible with existing rows? Can an index or backfill be rolled out without an unacceptable lock or outage? A `down()` method is not a complete recovery plan for a destructive migration. For related writes, consider transactions and database constraints, especially when requests can race.

For queued work, ask what happens on timeout, retry and permanent failure. Could retrying send a duplicate message or charge a customer twice? Use a domain-appropriate idempotency strategy and test it. A job that needs records written in a transaction should not run before those records are committed; Laravel documents dispatching after commit. That timing alone does not make an external side effect exactly once. [4]

Give feedback the author can use

Separate a merge blocker from a preference. Explain the observable risk and leave room for the author to propose a fix. For example:

> **Required:** Could we scope this invoice lookup to the user's school and add a cross-school feature test? As written, another school's invoice ID may reach this update path.

> **Optional:** Would a descriptive method name make this branch easier to follow? The current behaviour is correct, so this can wait.

These are example comments, not quotes from an actual review. Google's review guidance recommends courtesy, explaining reasoning and labelling severity. [6] If the author is unfamiliar with policies or the domain, offer to pair on the first change. Acknowledge a good test or a clear explanation too, with a reason why it helped.

Before approval, check the requested fixes and relevant tests again. Automated formatting and static checks can handle repetitive issues, leaving review time for behaviour and risk. Approval is a shared decision about the change, not a judgement of the developer.

*[Figure: image]***Multi-tenant Laravel SaaS: the architecture I actually use**: Single database or one per tenant? A production-tested multi-tenant setup for Laravel SaaS: global scopes, tenant resolution, queues and the mistakes to avoid. *[Figure: image]***Shipping AI features in Laravel without blocking your app**: LLM calls are slow, flaky and expensive. How I wrap them in Laravel with queues, streaming, retries, caching and cost tracking so users never wait on a spinner.

> **Key takeaways**
>
> - Understand the user outcome before reviewing implementation details.
> - Check authorisation and tenant boundaries independently of validation.
> - Review database changes and job retries as production behaviour.
> - Test denied access and unchanged state, not only the successful response.
> - Explain blockers respectfully and label optional suggestions.

Questions readers ask

What should a Laravel code review check first?

Understand the intended behaviour, then check authentication, authorisation and tenant boundaries before moving to validation, data integrity and performance. Review tests and deployment risks before approval.

Is request validation enough to secure a Laravel endpoint?

No. Validation checks the shape and constraints of input; an endpoint also needs appropriate authentication, resource-level authorisation and tenant or ownership checks.

How should I give feedback to a junior developer?

Describe the issue in the code, explain its impact and label whether a change is required or optional. Ask for a test or offer to pair when that would help the author learn.

References

1. Laravel. Authorization.
2. Laravel. Validation.
3. Laravel. Eloquent: Relationships.
4. Laravel. Queues.
5. Laravel. Database Testing.
6. Google. How to write code review comments.

CITE AS

Gulzaib, M. (2026, May 18). My code review checklist for junior Laravel developers. Gul’s Notebook. https://gulzaib.vercel.app/blog/my-code-review-checklist-for-junior-laravel-developers

You may share and adapt this text for any purpose, including commercially, as long as you credit the author and link to the original. Licence: https://creativecommons.org/licenses/by/4.0/