GUL'S NOTEBOOK · ENGINEERING · OPEN ACCESS

Multi-tenant Laravel SaaS: the architecture I actually use

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

Published	2026-08-10 (updated 2026-11-02)
URL	https://gulzaib.vercel.app/blog/multi-tenant-laravel-saas-the-architecture-i-actually-use
Licence	Creative Commons Attribution 4.0 International (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

ABSTRACT

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.

Keywords: laravel, saas, architecture, php, multi-tenancy

> **TL;DR**
>
> For most early SaaS products I use **one database with a `tenant_id` column**, enforced by a global scope and a tenant context resolved from the subdomain. It is simple to run, cheap to host and easy to migrate later. Below is the full setup, plus when I would choose a database per tenant instead.

Every SaaS I've built, from the POS to Oxygen LMS, has had to answer the same question on day one: how do we keep each customer's data separate? Get it wrong and you either leak data between customers or drown in operational complexity.

The three options

Strategy	Isolation	Ops cost	Good for
Shared DB, `tenant_id` column	Logical	Low	Most early-stage SaaS
Schema per tenant	Medium	Medium	Postgres shops, moderate tenant counts
Database per tenant	Strong	High	Enterprise, compliance, noisy neighbours

Decision · Accepted

Start with a shared database and a tenant_id column

Context. Small team, hundreds of tenants expected, low budget, fast iteration needed.

Decision. One MySQL database; every tenant-owned table has an indexed `tenant_id`; isolation enforced in the application layer with a global scope.

Consequences. Cheap and simple. The cost is discipline: every query must be scoped, so we enforce it automatically and test it.

Project layout

- app/
  - **Tenancy/**
    - TenantContext.php
    - BelongsToTenant.php
    - TenantScope.php
  - Http/Middleware/
    - **ResolveTenant.php**
  - Models/
    - Tenant.php
    - Student.php
- database/migrations/
- tests/Feature/
  - **TenantIsolationTest.php**

Step by step

1. **Hold the current tenant in one place**

   A tiny singleton that knows who the current tenant is. Everything else asks it.

   `app/Tenancy/TenantContext.php`

```
final class TenantContext
{
    private ?Tenant $tenant = null;

    public function set(Tenant $tenant): void { $this->tenant = $tenant; }

    public function get(): ?Tenant { return $this->tenant; }
```

```
    public function id(): ?int { return $this->tenant?->id; }
}
```

Register it as a singleton in a service provider: `$this->app->singleton(TenantContext::class);`

2. **Resolve the tenant from the request**

   I use subdomains (`school-a.oxygenlms.com`). Resolve once per request in middleware and fail closed if the tenant is unknown.

   `app/Http/Middleware/ResolveTenant.php`

```
public function handle(Request $request, Closure $next)
{
    $sub = explode('.', $request->getHost())[0];

    $tenant = Tenant::query()
        ->where('subdomain', $sub)
        ->where('active', true)
        ->firstOrFail(); // fail closed: never fall back to "no tenant"

    app(TenantContext::class)->set($tenant);

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

3. **Scope every query automatically**

   A global scope plus a trait means developers can't forget the `where tenant_id = ?` clause.

   BelongsToTenant.php

```
trait BelongsToTenant
{
    protected static function bootBelongsToTenant(): void
    {
        static::addGlobalScope(new TenantScope);

        static::creating(function ($model) {
            $model->tenant_id ??= app(TenantContext::class)->id()
                ?? throw new LogicException('No tenant in context');
        });
    }
}
```

   TenantScope.php

```
final class TenantScope implements Scope
{
    public function apply(Builder $builder, Model $model): void
    {
        $id = app(TenantContext::class)->id();

        // Fail closed: no tenant means no rows, never all rows.
        $builder->where($model->qualifyColumn('tenant_id'), $id ?? 0);
    }
}
```

Student.php

```
class Student extends Model
{
    use BelongsToTenant;
}
```

4. **Carry the tenant into queued jobs**

   Queues are where tenancy bugs hide. A job runs in a different process, so the context is empty. Store the tenant ID on the job and restore it before `handle()`.

   `app/Jobs/SendFeeReminders.php`

```
class SendFeeReminders implements ShouldQueue
{
    public function __construct(public int $tenantId) {}

    public function handle(TenantContext $ctx): void
    {
        $ctx->set(Tenant::findOrFail($this->tenantId));
        // ... scoped queries are safe from here
    }
}
```

5. **Prove isolation with a test**

   This is the most important test in the codebase. It runs on every push.

   `tests/Feature/TenantIsolationTest.php`

```
it('never returns another tenant's students', function () {
    [$a, $b] = Tenant::factory()->count(2)->create();
    Student::factory()->for($b)->create();

    app(TenantContext::class)->set($a);

    expect(Student::count())->toBe(0);
});
```

> **Watch out for raw queries**
>
> Global scopes don't apply to `DB::table()` or raw SQL. Ban them for tenant tables in code review, or wrap them in a helper that adds the tenant clause.

Indexes matter more than you think

With a shared database, almost every query filters by `tenant_id` first. Put it at the front of your composite indexes:

```
-- good: matches "where tenant_id = ? and class_id = ? order by name"
CREATE INDEX students_tenant_class_name ON students (tenant_id, class_id, name);
```

When I'd choose a database per tenant

Signals that it's time to move to stronger isolation

- A customer contract requires physical data separation or data residency.
- One tenant is so large that it slows everyone else down (noisy neighbour).
- You need per-tenant backups and restores as a product feature.

Because every model already carries `tenant_id`, moving a large tenant to its own database later is a data migration, not a rewrite.

> **Key takeaways**
>
> - Start simple: one database, a `tenant_id` column, a global scope.
> - Fail closed everywhere: no tenant means no data.
> - Queues and raw queries are where leaks happen. Test them.
> - Put `tenant_id` first in composite indexes.

> Update · 2 Nov 2026
>
> Added the queued jobs section after a reader asked how the context survives in workers.

Questions readers ask

Should a Laravel SaaS use one database or a database per tenant?

For most early SaaS products, one database with a tenant_id column enforced by a global scope is simpler to run, cheaper to host and easy to migrate later.

When should you move to a database per tenant?

When a contract requires physical separation or data residency, when one tenant is large enough to slow everyone else down, or when per-tenant backups and restores are a product

feature.

Where do multi-tenant data leaks usually happen?

In queued jobs and raw queries, where the global scope or tenant context may be missing. Fail closed, so no tenant means no data, and test those paths explicitly.

CITE AS

Gulzaib, M. (2026, August 10). Multi-tenant Laravel SaaS: the architecture I actually use. Gul's Notebook. https://gulzaib.vercel.app/blog/multi-tenant-laravel-saas-the-architecture-i-actually-use

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/