1v0 Ship v1.0 Faster

Using Laravel Factories and Seeders for Test Data

Populating reliable test and demo data is essential for development speed, realistic QA, and repeatable CI. In this guide, you'll learn what **factories** are (and why they matter), how to build **seeders** that are safe and idempotent, how to scale with an **extended folder structure** for large projects, and how to tailor seeding for **environments** (local/staging/production) and even **multi-tenant** apps.

What Are Factories (and Why Use Them)?

Factories generate model instances with realistic attributes for testing, seeding, and demos. They keep your code DRY, encourage consistent data shapes, and make complex relationship graphs trivial to build. Instead of hand-writing arrays, you call factory builders that know how to produce valid models (with relationships, states, and sequences).

Basic Factory Definition

```
// database/factories/UserFactory.php
namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    protected $model = User::class;

    public function definition(): array
    {
        return [
```

```
            'name' => $this->faker->name(),
            'email' => $this->faker->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => bcrypt('password'), // don't use in
production
            'remember_token' => Str::random(10),
        ];
    }
}Code language: PHP (php)
```

This factory tells Laravel how to build a valid `User` with realistic defaults. Use it for tests (`User::factory()`) and also inside seeders to generate bulk data.

Factory States (Variants)

```
// database/factories/UserFactory.php (continued)
public function admin(): static
{
    return $this->state(fn () => [
        'is_admin' => true,
    ]);
}

public function unverified(): static
{
    return $this->state(fn () => [
        'email_verified_at' => null,
    ]);
}Code language: PHP (php)
```

States let you quickly switch variants like admin users or unverified users: `User::factory()->admin()->create()`. Combine states freely.

Sequences (Patterned Data)

```
User::factory()
    ->count(4)
    ->sequence(
        ['plan' => 'free'],
```

```php
        ['plan' => 'pro'],
        ['plan' => 'business'],
        ['plan' => 'enterprise'],
    )
    ->create();
```
Code language: PHP (php)

Sequences rotate attribute sets per model, great for realistic mixes (pricing tiers, locales, statuses).

Relationships (for / has / hasAttached)

```php
// Example: A Post belongs to a User and has many Tags via pivot
use App\Models\Post;
use App\Models\Tag;

Post::factory()
    ->for(User::factory()->admin(), 'author')
    ->hasAttached(Tag::factory()->count(3), ['weight' => 1]) // pivot fields
    ->count(10)
    ->create();
```
Code language: PHP (php)

`for()` assigns a parent relation; `has()`/`hasAttached()` creates children (including pivot attributes). This builds correct graphs with one call.

Seeders: Repeatable, Idempotent Data

Seeders populate databases with base lookups, reference data, demo content, and admin accounts. Make them *idempotent* (safe to run multiple times) so deploys and CI stay reliable.

DatabaseSeeder as the Orchestrator

```php
// database/seeders/DatabaseSeeder.php
namespace Database\Seeders;

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call([
            LookupSeeder::class,
            AdminUserSeeder::class,
            DemoContentSeeder::class,
        ]);
    }
}
```
Code language: PHP (php)

The root seeder calls sub-seeders in a controlled order. Keep "base system" seed separate from "demo" seed for clarity.

Idempotent Patterns (firstOrCreate / upsert)

```php
// database/seeders/LookupSeeder.php
use App\Models\Plan;
use Illuminate\Database\Seeder;

class LookupSeeder extends Seeder
{
    public function run(): void
    {
        Plan::upsert([
            ['code' => 'free', 'name' => 'Free', 'price' => 0],
            ['code' => 'pro', 'name' => 'Pro', 'price' => 19],
        ], ['code'], ['name','price']);
    }
}
```
Code language: PHP (php)

`upsert` updates or inserts by a unique key; `firstOrCreate` is perfect for single rows. This keeps seeders re-runnable.

Model::unguard + FK Handling (MySQL)

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\DB;

// inside a seeder run():
Model::unguard();
DB::statement('SET FOREIGN_KEY_CHECKS=0');
// ... perform bulk inserts / truncations cautiously ...
DB::statement('SET FOREIGN_KEY_CHECKS=1');
Model::reguard();
```
Code language: PHP (php)

Temporarily unguard models for bulk inserts and use FK toggles carefully when resetting demo data. Avoid in production seed unless you know the implications.

Extended Folder Structure for Many Seeders

Large apps benefit from grouping seeders by domain. Keep “core” vs “demo” clear, and isolate third-party integration seeds (e.g., roles/permissions) for repeatability.

```
database/
└─ seeders/
   ├─ Core/
   │  ├─ Lookup/
   │  │  ├─ CountriesSeeder.php
   │  │  └─ PlansSeeder.php
   │  ├─ Auth/
   │  │  ├─ AdminUserSeeder.php
   │  │  └─ RolesPermissionsSeeder.php
   │  └─ Content/
```

```
│       └─ SettingsSeeder.php
├─ Demo/
│   ├─ UsersDemoSeeder.php
│   ├─ BlogDemoSeeder.php
│   └─ OrdersDemoSeeder.php
└─ Tenants/
    ├─ TenantBootstrapSeeder.php
    └─ TenantDemoSeeder.phpCode language: Bash (bash)
```

This structure scales: `Core` is minimal system data; `Demo` is optional; `Tenants` handles multi-tenant bootstrap. PSR-4 autoloading works for nested namespaces under `Database\Seeders`.

Namespacing & Calling Sub-Seeders

```
// database/seeders/DatabaseSeeder.php
namespace Database\Seeders;

use Database\Seeders\Core\Lookup\CountriesSeeder;
use Database\Seeders\Core\Lookup\PlansSeeder;
use Database\Seeders\Core\Auth\AdminUserSeeder;
use Database\Seeders\Core\Auth\RolesPermissionsSeeder;
use Database\Seeders\Demo\BlogDemoSeeder;

class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        $this->call([
            CountriesSeeder::class,
            PlansSeeder::class,
            RolesPermissionsSeeder::class,
            AdminUserSeeder::class,
        ]);

        if (app()->environment(['local','staging'])) {
            $this->call([BlogDemoSeeder::class]);
        }
    }
```

```
}
```
Code language: PHP (php)

Keep imports explicit. Use environment checks to include demo-only data locally while keeping production clean.

Environment & Tenant-Aware Seeding

Environment-Specific

```bash
# run only demo seeds locally
php artisan db:seed --class=Database\\Seeders\\Demo\\BlogDemoSeeder
```
Code language: Bash (bash)

Use command-line control for targeted seeds. Inside `DatabaseSeeder`, conditionally call demo seeders with `app()->environment([...])`.

Per-Tenant (Loop & Switch)

```php
// database/seeders/Tenants/TenantBootstrapSeeder.php
use App\Models\Tenant;
use Illuminate\Support\Facades\DB;

class TenantBootstrapSeeder extends Seeder
{
    public function run(): void
    {
        foreach (Tenant::cursor() as $tenant) {
            // switch connection dynamically, then seed
            DB::setDefaultConnection($tenant->connection);
            $this->call([
                TenantSchemaSeeder::class,
                TenantAdminSeeder::class,
            ]);
```

```
        }
    }
}
```
Code language: PHP (php)

Multi-tenant apps often seed per-tenant databases. Loop tenants, switch connection (or use a tenancy package helper), and call tenant-specific seeders.

Generating Bulk Data with Factories

```
// database/seeders/Demo/BlogDemoSeeder.php
use App\Models\User;
use App\Models\Post;
use App\Models\Tag;

class BlogDemoSeeder extends Seeder
{
    public function run(): void
    {
        $users = User::factory()->count(10)->create();

        $tags = Tag::factory()
            ->count(8)
            ->sequence(fn ($sequence) => ['weight' => $sequence->index
+ 1])
            ->create();

        Post::factory()
            ->count(50)
            ->for($users->random(), 'author')
            ->hasAttached($tags->random(rand(2,4)))
            ->create();
    }
}
```
Code language: PHP (php)

Factories make it trivial to create consistent, high-volume demo data with correct relations. Use `sequence` for predictable patterns and `random()` for variety.

Performance Tips

```
// Example: bulk inserts with minimal event noise
Post::withoutEvents(function () {
    Post::factory()->count(2000)->create();
});
```
Code language: PHP (php)

Wrap heavy factory runs with `withoutEvents` if event/listener overhead is large. Consider chunking tasks, disabling observers, and using optimized indexes.

Using Factories Directly in Tests

```
// tests/Feature/PostApiTest.php
use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use App\Models\User;
use App\Models\Post;

class PostApiTest extends TestCase
{
    use RefreshDatabase;

    public function test_index_requires_auth(): void
    {
        $this->getJson('/api/posts')->assertUnauthorized();
    }

    public function test_index_returns_posts(): void
    {
        $user = User::factory()->create();
```

```
        Post::factory()->count(3)->for($user, 'author')->create();

        $this->actingAs($user)->getJson('/api/posts')
            ->assertOk()
            ->assertJsonCount(3, 'data');
    }
}
```
Code language: PHP (php)

Factories shine in tests: realistic data, clearer intent, and isolated states per test using `RefreshDatabase`.

Optional: Admin-Only UI to Run Demo Seeds

```
// routes/web.php
Route::post('/admin/run-demo-seed', [AdminSeedController::class,
'run'])
    ->middleware(['auth','can:admin']);

// app/Http/Controllers/AdminSeedController.php
use Illuminate\Support\Facades\Artisan;

class AdminSeedController
{
    public function run()
    {
        Artisan::call('db:seed', [
            '--class' => 'Database\\Seeders\\Demo\\BlogDemoSeeder',
            '--force' => true,
        ]);

        return back()->with('status','Demo data seeded.');
    }
}
```
Code language: PHP (php)

Expose a safe, admin-guarded endpoint to trigger demo seeds for staging or QA. Never allow anonymous access to seed endpoints.

Wrapping Up

Factories produce realistic models with minimal code; seeders assemble them into coherent datasets for base lookups, demos, and tests. With a scalable folder structure, idempotent patterns (`upsert`/`firstOrCreate`), and environment/tenant awareness, your data pipeline stays fast, reliable, and CI-friendly.

What's Next

Level up your testing and data quality with these:

- Testing Laravel Applications with PHPUnit
- How to Write Feature Tests in Laravel for APIs
- Laravel Eloquent Relationships Explained with Examples