1v0 Ship v1.0 Faster

How to Build Email Verification in Laravel 12 (Step by Step)

One of the first security features you should add to any web app is **Email Verification**. When a new user signs up, you want to be sure the email they provided is valid and belongs to them. Without verification, users could register with fake or disposable emails, which leads to spam, weak security, and a lack of trust in your application.

In this step-by-step tutorial, we'll build a complete **Email Verification system in Laravel 12**. We'll explain not just the code, but also why each step matters, so even beginners can follow along. By the end, your app will send verification emails, require verified emails for protected pages, and provide a simple UI for resending verification links.

1 – What is Email Verification?

Email Verification means a new user must confirm ownership of their email before they can fully use your app. The system sends a unique, signed link to their inbox. Clicking the link marks their account as verified. This prevents fake accounts, strengthens security, and ensures you can reach users reliably (for password resets, notifications, etc.).

Laravel 12 includes built-in verification support. We'll turn it on, add routes and UI, and protect pages with the `verified` middleware.

2 – Prerequisites

- Laravel 12 project (PHP 8.2+) with a working database connection in `.env`
- Basic auth (register & login) already in place (can be hand-rolled or any scaffolding)
- Mail configured in `.env` (SMTP/Mailgun/Sendgrid/etc.) so the app can send emails

3 – Enable Email Verification on the User model

Add the `MustVerifyEmail` interface to your `User` model. This tells Laravel that users must verify their email before accessing routes protected by the `verified` middleware.

```
// app/Models/User.php
namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail; // <-- add this
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class User extends Authenticatable implements MustVerifyEmail
{
    use HasFactory, Notifiable;

    // fillable/hidden/casts as you normally have them
}
Code language: PHP (php)
```

Explanation: implementing `MustVerifyEmail` enables a few framework behaviors: Laravel knows how to send verification notifications, and the `verified` middleware will check `$user->hasVerifiedEmail()` for protected routes.

4 – Add the verification routes (notice, verify, resend)

Create three routes: a “please verify your email” page, the *confirmation* endpoint (clicked from email), and a *resend* endpoint (rate-limited) in case the user didn’t receive the email.

```
// routes/web.php
use Illuminate\Foundation\Auth\EmailVerificationRequest;
use Illuminate\Http\Request;

// 1) Notice page (unverified users land here)
Route::get('/email/verify', function () {
    return view('auth.verify-email'); // we'll create this view next
})->middleware('auth')->name('verification.notice');

// 2) Verification callback (user clicks link in email)
Route::get('/email/verify/{id}/{hash}', function
(EmailVerificationRequest $request) {
    $request->fulfill(); // marks the user's email as verified
    return redirect('/dashboard'); // or wherever you want verified
users to land
})->middleware(['auth', 'signed'])->name('verification.verify');

// 3) Resend verification link (rate-limited)
Route::post('/email/verification-notification', function (Request
$request) {
    $request->user()->sendEmailVerificationNotification();
    return back()->with('status', 'Verification link sent!');
})->middleware(['auth', 'throttle:6,1'])->name('verification.send');
Code language: PHP (php)
```

Explanation: The `signed` middleware ensures the verification link has not been tampered with. The `throttle:6,1` middleware lets users request a new link but prevents abuse (max

6 per minute).

5 – Build the “Verify your email” Blade view (with resend button)

Create a friendly, minimal UI that explains why verification is needed and allows resending the email.

```
<!-- resources/views/auth/verify-email.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container py-5" style="max-width:720px">
  <h1 class="h4 mb-3">Verify your email address</h1>

  <p>Thanks for signing up! Before getting started, please verify your email by clicking the link we just sent you.</p>

  @if (session('status') === 'verification-link-sent')
    <div class="alert alert-success mt-3">
      A new verification link has been sent to your email address.
    </div>
  @endif

  <form method="POST" action="{{ route('verification.send') }}" class="mt-3">
    @csrf
    <button class="btn btn-primary">Resend verification email</button>
  </form>
```

```
  <p class="text-muted small mt-3 mb-0">
    Didn’t get the email? Check spam, or click the button above to
resend.
  </p>
</div>
@endsection
Code language: PHP (php)
```

Explanation: We use a neutral success message so users don’t wonder if the button “worked.” The message displays whenever a new link is sent.

6 – Protect pages with the `verified` middleware

Add the `verified` middleware to any route that should only be accessible after email confirmation. Unverified users are redirected to the notice page.

```
// routes/web.php
Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified']);
Code language: PHP (php)
```

Explanation: The middleware calls `$request->user()->hasVerifiedEmail()`. If it returns `false`, the user is redirected to `verification.notice` (`/email/verify`).

7 – Configure mail so emails actually send

Make sure your `.env` is configured, otherwise emails will silently fail or go nowhere.

```
# .env (example SMTP settings)
MAIL_MAILER=smtp
MAIL_HOST=smtp.your-provider.com
MAIL_PORT=587
MAIL_USERNAME=your_user
MAIL_PASSWORD=your_password
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=no-reply@your-domain.com
MAIL_FROM_NAME="${APP_NAME}"Code language: PHP (php)
```

Explanation: Test quickly by registering a new account in local/dev. If you don't want to send real emails locally, use a mail catcher (MailHog/HELO) and point SMTP to it.

8 – (Optional) Customize the verification email

Laravel uses a notification to send the verification email. You can customize the email by overriding the notification's `toMail` on a `VerifyEmail` class and registering it, or by customizing mail styling with Markdown mail templates. A lightweight approach is to customize `MAIL_FROM_*` and branding in your layout so it aligns with your app's voice.

(If you later need a branded template, create a custom notification that extends the default and uses your Markdown view. Keep the signed URL intact for security.)

9 – Test the entire flow (step-by-step)

1. Register a new user (ensure mail settings are correct).
2. Try visiting a protected page (like `/dashboard`) — you should be redirected to `/email/verify`.
3. Open your inbox and click the verification link.
4. You should be redirected and now able to access `/dashboard`.
5. Try the “Resend verification email” button if needed.

10 – Common errors & fixes

- **Verification email not received:** Double-check `.env` SMTP credentials and logs. Use a mail catcher locally. Check spam folders in staging/production.
- **“Invalid signature” error:** The verification link was modified or expired. Ask the user to resend the link from the verify page.
- **Still redirected to verify after clicking link:** Ensure the `User` model implements `MustVerifyEmail`, and your protected routes include the `verified` middleware.
- **Local dev with tunnels:** If using a tunnel (e.g., ngrok), ensure the URL in the email matches the environment you’re testing in.

Wrapping Up

You've built a complete **Email Verification flow in Laravel 12**: enabling verification on the `User` model, adding secure routes, creating a clean UI with a resend button, and protecting sensitive pages with the `verified` middleware. This simple feature dramatically improves trust and safety in your application.

What's Next

- Implementing Two-Factor Authentication in Laravel — add another layer of login protection.
- How to Restrict Page Access by Role in Laravel 12 — ensure only the right users see the right pages.
- Laravel Authentication: Redirect Users After Login & Logout — polish the user journey.