1v0 Ship v1.0 Faster

How to Use Eloquent Events for Auditing User Actions

How to Use Eloquent Events for Auditing User Actions

Auditing records who did what and when—essential for debugging, compliance, and customer support. With Eloquent events (creating, created, updating, updated, deleting, deleted, restored, forceDeleted), we can capture old/new values, the actor, and request metadata into an *audits* table. In this guide you'll build a reusable observer, register it, and ship a simple UI to inspect audit trails.

1 – Migration: Audits Table

Create a generic table to store audit entries for any model using a polymorphic relation. We'll store the event name, old/new values, and request context.

```
// database/migrations/2025_08_27_000000_create_audits_table.php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void {
        Schema::create('audits', function (Blueprint $table) {
            $table->id();
            // polymorphic: auditable_type + auditable_id
            $table->morphs('auditable');
            // actor (nullable for system jobs)
$table->foreignId('user_id')->nullable()->constrained()->nullOnDelete(
);
            // event: created, updated, deleted, restored,
force_deleted
```

```
            $table->string('event', 32);
            // JSON diffs
            $table->json('old_values')->nullable();
            $table->json('new_values')->nullable();
            // request context
            $table->string('ip_address', 45)->nullable();
            $table->text('user_agent')->nullable();
            $table->timestamps();
            $table->index(['event', 'created_at']);
        });
    }

    public function down(): void {
        Schema::dropIfExists('audits');
    }
};Code language: PHP (php)
```

This table can attach audits to any model via `morphs('auditable')`. We keep `old_values`/`new_values` as JSON to store only changed keys for updates and full snapshots for create/delete if you prefer.

2 – Audit Model

Define the `Audit` model with casts and the polymorphic relation back to the auditable model.

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

use Illuminate\Database\Eloquent\Model;

class Audit extends Model
{
```

```
    protected $fillable = [
        'auditable_type','auditable_id','user_id','event',
        'old_values','new_values','ip_address','user_agent'
    ];

    protected $casts = [
        'old_values' => 'array',
        'new_values' => 'array',
    ];

    public function auditable()
    {
        return $this->morphTo();
    }

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}Code language: PHP (php)
```

Casting JSON arrays makes reading diffs straightforward in controllers and Blade. The `auditable()` morph lets you navigate back to the source record.

3 – Reusable Observer to Capture Events

The observer listens to Eloquent lifecycle events, computes diffs, redacts sensitive fields, and writes an `Audit` entry. Attach the same observer to any model you want to audit.

```
// app/Observers/GenericAuditObserver.php
namespace App\Observers;

use App\Models\Audit;
```

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Arr;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Request;

class GenericAuditObserver
{
    /**
     * Keys to hide in audits (passwords, tokens, secrets).
     */
    protected array $redacted =
['password','remember_token','api_key','secret','token'];

    protected function redact(array $data): array
    {
        foreach ($this->redacted as $key) {
            if (Arr::has($data, $key)) {
                Arr::set($data, $key, '***redacted***');
            }
        }
        return $data;
    }

    protected function context(): array
    {
        return [
            'user_id'    => Auth::id(),
            'ip_address' => Request::ip(),
            'user_agent' => Request::header('User-Agent'),
        ];
    }

    public function created(Model $model): void
    {
        Audit::create([
            'auditable_type' => get_class($model),
            'auditable_id'   => $model->getKey(),
            'event'          => 'created',
```

```
            'old_values'     => null,
            'new_values'     =>
$this->redact($model->getAttributes()),
        ] + $this->context());
    }

    public function updated(Model $model): void
    {
        // changed attributes only
        $changes = $model->getChanges();             // new values for
dirty fields
        $original = Arr::only($model->getOriginal(),
array_keys($changes));

        Audit::create([
            'auditable_type' => get_class($model),
            'auditable_id'   => $model->getKey(),
            'event'          => 'updated',
            'old_values'     => $this->redact($original),
            'new_values'     => $this->redact($changes),
        ] + $this->context());
    }

    public function deleted(Model $model): void
    {
        Audit::create([
            'auditable_type' => get_class($model),
            'auditable_id'   => $model->getKey(),
            'event'          => 'deleted',
            'old_values'     =>
$this->redact($model->getAttributes()),
            'new_values'     => null,
        ] + $this->context());
    }

    public function restored(Model $model): void
    {
        Audit::create([
```

```
            'auditable_type' => get_class($model),
            'auditable_id'   => $model->getKey(),
            'event'          => 'restored',
            'old_values'     => null,
            'new_values'     => null,
        ] + $this->context());
    }

    public function forceDeleted(Model $model): void
    {
        Audit::create([
            'auditable_type' => get_class($model),
            'auditable_id'   => $model->getKey(),
            'event'          => 'force_deleted',
            'old_values'     => null,
            'new_values'     => null,
        ] + $this->context());
    }
}Code language: PHP (php)
```

The observer captures five major events. For updates, it stores only changed keys by diffing `getOriginal()` vs `getChanges()`. The `$redacted` list ensures secrets never leak into the audit log.

4 – Register the Observer

You can observe multiple models. Here we audit `User` and `Post`. Add more as needed.

```
// app/Providers/AppServiceProvider.php (boot)
use App\Models\Post;
use App\Models\User;
use App\Observers\GenericAuditObserver;
```

```
public function boot(): void
{
    User::observe(GenericAuditObserver::class);
    Post::observe(GenericAuditObserver::class);
}Code language: PHP (php)
```

Placing this in `AppServiceProvider::boot()` wires the observer globally. From now on, creates/updates/deletes on these models produce audit rows automatically.

5 – (Optional) Auditable Trait for Model-Side Opt-In

If you prefer models to opt-in explicitly, use a trait that registers the observer on boot. This avoids listing models in the provider.

```
// app/Models/Concerns/Auditable.php
namespace App\Models\Concerns;

use App\Observers\GenericAuditObserver;

trait Auditable
{
    public static function bootAuditable(): void
    {
        static::observe(GenericAuditObserver::class);
    }
}Code language: PHP (php)
```

Attach the trait to any model you want audited: `use Auditable;`. When the model boots, it registers the observer automatically.

6 – Querying the Audit Log

A controller to filter audits by model, event, date range, or user. We'll display results in a simple Blade table.

```
// app/Http/Controllers/AuditController.php
namespace App\Http\Controllers;

use App\Models\Audit;
use Illuminate\Http\Request;

class AuditController extends Controller
{
    public function index(Request $request)
    {
        $q = Audit::query()
            ->with(['user'])
            ->latest();

        if ($m = $request->input('model')) {
            // expects fully-qualified class or short name
            $q->where('auditable_type', $m);
        }

        if ($e = $request->input('event')) {
            $q->where('event', $e);
        }

        if ($u = $request->input('user_id')) {
            $q->where('user_id', $u);
        }
```

```
        if ($from = $request->date('from')) {
            $q->whereDate('created_at', '>=', $from);
        }
        if ($to = $request->date('to')) {
            $q->whereDate('created_at', '<=', $to);
        }

        $audits = $q->paginate(15)->withQueryString();
        return view('audits.index', compact('audits'));
    }
}Code language: PHP (php)
```

The controller builds a flexible query over the `audits` table. We eager-load the actor (`user`) and support common filters to narrow results.

```
// routes/web.php (snippet)
use App\Http\Controllers\AuditController;

Route::middleware(['auth'])->group(function () {
    Route::get('/audits', [AuditController::class,
'index'])->name('audits.index');
});Code language: PHP (php)
```

Audit visibility should be restricted—only admins or support roles should access this page. Protect the route with `auth` and authorization policies as needed.

7 – UI: Audit Log Blade with Filters

A minimal interface to explore audit entries, see who changed what, and inspect JSON diffs quickly.

```
<!-- resources/views/audits/index.blade.php -->
@extends('layouts.app')
```

```
@section('content')
<div class="container">
  <h1 class="mb-4">Audit Log</h1>

  <form method="GET" class="row g-2 mb-4">
    <div class="col-md-3">
      <input name="model" class="form-control" placeholder="Model
(FQCN)" value="{{ request('model') }}" />
    </div>
    <div class="col-md-2">
      <select name="event" class="form-select">
        @php $events =
['created','updated','deleted','restored','force_deleted']; @endphp
        <option value="">Any event</option>
        @foreach($events as $e)
          <option value="{{ $e }}" {{ request('event')===$e ?
'selected' : '' }}>{{ ucfirst(str_replace('_',' ',$e)) }}</option>
        @endforeach
      </select>
    </div>
    <div class="col-md-2">
      <input type="number" name="user_id" class="form-control"
placeholder="User ID" value="{{ request('user_id') }}" />
    </div>
    <div class="col-md-2">
      <input type="date" name="from" class="form-control" value="{{
request('from') }}" />
    </div>
    <div class="col-md-2">
      <input type="date" name="to" class="form-control" value="{{
request('to') }}" />
    </div>
    <div class="col-md-1 d-grid">
      <button class="btn btn-theme">Filter</button>
    </div>
  </form>

  @forelse($audits as $audit)
```

```
    <div class="card mb-3">
      <div class="card-body">
        <div class="d-flex justify-content-between align-items-
center">
          <div>
            <strong>{{ class_basename($audit->auditable_type) }}#{{
$audit->auditable_id }}</strong>
            <span class="badge bg-secondary">{{ $audit->event
}}</span>
          </div>
          <small class="text-muted">{{ $audit->created_at }} by {{
optional($audit->user)->name ?? 'system' }}</small>
        </div>

        <div class="mt-3">
          <details>
            <summary>Old Values</summary>
            <pre class="mb-0">{{ json_encode($audit->old_values,
JSON_PRETTY_PRINT|JSON_UNESCAPED_SLASHES) }}</pre>
          </details>
          <details class="mt-2">
            <summary>New Values</summary>
            <pre class="mb-0">{{ json_encode($audit->new_values,
JSON_PRETTY_PRINT|JSON_UNESCAPED_SLASHES) }}</pre>
          </details>
        </div>

        <small class="text-muted">IP: {{ $audit->ip_address }} |
Agent: {{ Str::limit($audit->user_agent, 80) }}</small>
      </div>
    </div>
  @empty
    <p class="text-muted">No audit entries found.</p>
  @endforelse

  {{ $audits->links() }}
</div>
@endsectionCode language: PHP (php)
```

The page shows each event with who did it and when. Expandable `<details>` blocks display JSON diffs without overwhelming the layout. Filters make it easy to narrow down incidents.

8 – Applying Auditing to a Model

Here's how you would opt-in a typical model (e.g., `Post`). We'll also show a quick controller action to trigger events.

```
// app/Models/Post.php (snippet)
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use App\Models\Concerns\Auditable;

class Post extends Model
{
    use SoftDeletes, Auditable;

    protected $fillable = ['user_id','title','body','status'];
}Code language: PHP (php)
```

Including `Auditable` ensures the observer is registered for `Post`. Combining with `SoftDeletes` records *deleted*, *restored*, and *force_deleted* events as well.

```
// app/Http/Controllers/PostController.php (snippet)
public function update(Request $request, Post $post)
{
    $data = $request->validate([
        'title' => ['required','string','max:150'],
        'body'  => ['required','string'],
        'status'=> ['required','in:draft,published'],
```

```
    ]);

    $post->update($data); // triggers "updated" audit
    return back()->with('status','Post updated.');
}Code language: PHP (php)
```

Saving the model is all that's needed—events fire automatically, the observer writes the audit, and you can view it in the Audit Log UI.

Wrapping Up

You built a robust audit trail with Eloquent events: a polymorphic audits table, a reusable observer that records diffs and request context, registration via provider or trait, and a filterable UI. This approach is lightweight, framework-native, and easy to extend with policies, retention rules, or offloading to a log index if volumes grow.

What's Next

- Eager Loading vs Lazy Loading in Laravel: Best Practices
- Filtering and Searching with Eloquent Query Builder
- 10 Proven Ways to Optimize Laravel for High Traffic