How to Build a Search Function in Laravel with Scout and Meilisearch

How to Build a Search Function in Laravel with Scout and Meilisearch

Modern applications need fast, typo-tolerant, and relevant search. Laravel provides **Scout**, a driver-based abstraction for full-text search engines, and **Meilisearch**, a blazing fast open-source search engine. Together, they make it simple to build advanced search into your Laravel app with minimal effort. In this guide, we'll explain what Scout and Meilisearch are, why they work well together, and how to set up a search feature with Blade UI and JSON API responses.

What is Laravel Scout?

Laravel Scout is an official Laravel package that provides a simple, driver-based abstraction for full-text search. Instead of writing raw queries against search engines, Scout lets you call methods like `Model::search()` directly on your Eloquent models. Behind the scenes, Scout syncs your models with the search index (Meilisearch, Algolia, Elasticsearch, etc.).

Why use it? Because Scout handles indexing, updating, and deleting records automatically. It integrates seamlessly with Eloquent, so you can focus on building features instead of writing search infrastructure code.

What is Meilisearch?

Meilisearch is an open-source, fast, and developer-friendly search engine. It provides features like typo tolerance, filters, relevancy ranking, and near-instant search results. It's lightweight, easy to host, and has great Laravel integration via Scout.

Why Meilisearch? Unlike SQL `LIKE` queries, Meilisearch is designed for real-time search experiences, supports autocomplete, and ranks results by relevancy rather than simple string matching. Perfect for blogs, e-commerce, and SaaS dashboards.

Why Use Scout with Meilisearch?

Scout acts as a bridge between Laravel and Meilisearch. Instead of learning Meilisearch's HTTP API directly, you work with clean Eloquent methods. Scout automatically syncs your models to Meilisearch whenever they're created, updated, or deleted.

- **Without Scout**: You would need to call Meilisearch's HTTP API manually for indexing and updates.
- **With Scout**: Just call `Post::search('query')` and get results instantly, with Eloquent models returned.

Install and Configure Scout + Meilisearch

```
composer require laravel/scout meilisearch/meilisearch-phpCode language: Bash (bash)
```

Publish the Scout config file:

```
php artisan vendor:publish --
provider="Laravel\Scout\ScoutServiceProvider"Code language: Bash (bash)
```

Update your `.env` file to use Meilisearch:

```
SCOUT_DRIVER=meilisearch
MEILISEARCH_HOST=http://127.0.0.1:7700
MEILISEARCH_KEY=masterKeyCode language: Bash (bash)
```

Run Meilisearch locally via Docker for development:

```
docker run -it --rm -p 7700:7700 getmeili/meilisearch:latestCode language: Bash (bash)
```

Prepare Your Model for Search

Add the `Searchable` trait to your Eloquent model. This tells Scout to sync it with Meilisearch.

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

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
```

```
    use Searchable;

    protected $fillable = ['title', 'content'];

    public function toSearchableArray(): array
    {
        return [
            'title' => $this->title,
            'content' => $this->content,
        ];
    }
}Code language: PHP (php)
```

Scout will automatically keep your search index in sync whenever a Post is created, updated, or deleted.

Build a Search Controller and Blade Form

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

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

class SearchController extends Controller
{
    public function index(Request $request)
    {
        $query = $request->input('q');
        $results = $query ? Post::search($query)->get() : collect();

        return view('search.index', compact('results', 'query'));
    }
```

}Code language: PHP (php)

```
<!-- resources/views/search/index.blade.php -->
<form method="GET" action="/search">
  <input type="text" name="q" value="{{ $query }}" placeholder="Search
posts...">
  <button type="submit">Search</button>
</form>

<ul>
  @foreach($results as $post)
    <li><a href="/posts/{{ $post->id }}">{{ $post->title }}</a></li>
  @endforeach
</ul>Code language: PHP (php)
```

This simple Blade template lets users search posts and displays matching results instantly from Meilisearch.

Expose a JSON API for Search

If you're building a Vue or React frontend, you may want search results as JSON. Add an API endpoint:

```
// routes/api.php
use App\Http\Controllers\SearchController;

Route::get('/search', [SearchController::class, 'api']);

// app/Http/Controllers/SearchController.php
public function api(Request $request)
{
    $query = $request->input('q');
    $results = $query ? Post::search($query)->paginate(10) :
```

```
collect();

    return response()->json($results);
}Code language: PHP (php)
```

Now, your JavaScript frontend can fetch `/api/search?q=keyword` and render results in real-time with autocomplete or infinite scroll.

Scout + Meilisearch vs SQL LIKE Queries

Feature	Scout + Meilisearch	Raw SQL LIKE
Speed	Fast, indexed search optimized for full-text	Slow for large datasets
Relevancy	Results ranked by relevance and typo tolerance	Basic string match only
Features	Autocomplete, filters, pagination, typo tolerance	No advanced features
Integration	Clean Eloquent `search()` API	Manual SQL queries
Scalability	Handles millions of records efficiently	Poor scalability

This table shows why using **Scout with Meilisearch** is the modern approach compared to raw SQL `LIKE` queries, especially for content-heavy applications.

Wrapping Up

In this article, you learned what **Laravel Scout** and **Meilisearch** are, why Scout is used as a bridge to Meilisearch, and how to build a search function in Laravel with both Blade and JSON API examples. Compared to SQL `LIKE` queries, Scout and Meilisearch deliver faster, smarter, and more scalable search results. This setup is perfect for blogs, SaaS apps, and e-

commerce platforms where speed and relevancy matter.

What’s Next

Keep building your Laravel search and SEO stack with these related guides:

- Implementing Full-Text Search in Laravel with MySQL
- Creating JSON-LD Structured Data in Laravel for SEO
- Laravel SEO Guide: Optimizing Meta, Slugs, and Sitemaps