Laravel SEO Guide: Optimizing Meta, Slugs, and Sitemaps

Laravel SEO Guide: Optimizing Meta, Slugs, and Sitemaps

SEO (Search Engine Optimization) is critical for visibility. Laravel doesn't ship with built-in SEO tools, but it gives you the flexibility to build them yourself. In this guide, you'll learn how to generate SEO-friendly slugs, handle duplicate slugs effectively, understand how `booted()` works, manage meta tags dynamically, and build a sitemap to help Google crawl your site efficiently.

Generating SEO-Friendly Slugs

URLs should be clean and keyword-rich. Laravel's `Str::slug()` helper converts titles into SEO-friendly slugs.

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

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Str;

class Post extends Model
{
    protected static function booted()
    {
        static::creating(function ($post) {
```

```
            $post->slug = Str::slug($post->title);
        });
    }
}Code language: PHP (php)
```

This automatically sets the slug when creating a new post. For example, “My First Blog Post” becomes `my-first-blog-post`. You can then use this slug in routes.

```
// routes/web.php
use App\Http\Controllers\PostController;

Route::get('/posts/{post:slug}', [PostController::class, 'show']);Code language: PHP (php)
```

Now users and search engines can access posts with clean URLs like `/posts/my-first-blog-post` instead of numeric IDs.

Performance Note: How `booted()` Works

The `booted()` method is often misunderstood. It does *not* run every time you initialize a model. Instead:

- **`booted()` runs once per request**, when Laravel loads the model class, to register event listeners like `creating` or `updating`.
- The closure you register (e.g., slug generation) runs **only when the event is fired** — in this case, whenever a new model is saved to the database.
- Simply fetching or instantiating a model (`Post::first()` or `new Post`) does **not** trigger slug generation.

This means the slug logic is only executed during `creating` events (new inserts), not for every model initialization or retrieval — keeping performance efficient.

Handling Duplicate Slugs (Optimized, Single Query Strategy)

If two posts share the same title, `Str::slug()` will generate identical slugs, leading to conflicts. To solve this efficiently, fetch the maximum numeric suffix in one query and append the next number.

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

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Str;

class Post extends Model
{
    protected static function booted()
    {
        static::creating(function ($post) {
            $base = Str::slug($post->title);

            if (! static::where('slug', $base)->exists()) {
                $post->slug = $base;
                return;
            }

            $pattern = '^' . preg_quote($base, '/') . '(-[0-9]+)?$';

            $maxSuffix = static::whereRaw('slug REGEXP ?', [$pattern])
                ->selectRaw("
                    MAX(
                        CASE
                            WHEN slug = ? THEN 0
                            ELSE CAST(SUBSTRING_INDEX(slug, '-', -1)
AS UNSIGNED)
                        END
                    ) AS max_suffix
                ", [$base])
                ->value('max_suffix');
```

```
            $next = ((int) $maxSuffix) + 1;
            $post->slug = "{$base}-{$next}";
        });
    }
}Code language: PHP (php)
```

This ensures slugs remain unique with minimal queries. For example, creating three posts with the same title results in:

- hello-world
- hello-world-1
- hello-world-2

Database Safety: Add a Unique Index

```
// database/migrations/xxxx_xx_xx_add_unique_index_to_posts_slug.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::table('posts', function (Blueprint $table) {
            $table->string('slug')->unique()->change();
        });
    }

    public function down(): void
    {
        Schema::table('posts', function (Blueprint $table) {
            $table->dropUnique(['slug']);
        });
    }
};Code language: PHP (php)
```

The unique index guarantees no duplicates even under concurrency. You can add retry logic on duplicate key errors if needed.

Portable Fallback (Any SQL)

```
$base = Str::slug($post->title);
$count = static::where('slug', 'like', $base.'%')->count();
$post->slug = $count ? "{$base}-{$count}" : $base;Code language: PHP (php)
```

This simpler fallback avoids loops and works on any database. It may skip gaps (e.g., jump to `hello-world-3` if `hello-world-2` was deleted), but that's perfectly fine for SEO.

Dynamic Meta Tags for SEO

Meta tags like `<title>`, `description`, and Open Graph tags improve click-through rates and rankings. Add them dynamically per page.

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

use App\Models\Post;

class PostController extends Controller
{
    public function show(Post $post)
    {
        return view('posts.show', compact('post'));
    }
}Code language: PHP (php)
```

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

@section('head')
  <title>{{ $post->title }} | My Blog</title>
  <meta name="description" content="{{ Str::limit($post->content, 160)
```

```
}}">

  <!-- Open Graph -->
  <meta property="og:title" content="{{ $post->title }}">
  <meta property="og:description" content="{{
Str::limit($post->content, 160) }}">
  <meta property="og:url" content="{{ url()->current() }}">
@endsection

@section('content')
  <h1>{{ $post->title }}</h1>
  <p>{{ $post->content }}</p>
@endsectionCode language: PHP (php)
```

UI Example: Adding Meta Fields

```
<form action="{{ route('posts.store') }}" method="POST">
  @csrf
  <label>Title</label>
  <input type="text" name="title">

  <label>Meta Description</label>
  <textarea name="meta_description"></textarea>

  <button type="submit">Save</button>
</form>Code language: PHP (php)
```

Editors can set meta descriptions manually. If empty, fall back to auto-generating from content.

Building an XML Sitemap

```
// routes/web.php
use App\Http\Controllers\SitemapController;
Route::get('/sitemap.xml', [SitemapController::class, 'index']);Code language: PHP (php)
```

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

use App\Models\Post;

class SitemapController extends Controller
{
    public function index()
    {
        $posts = Post::all();
        return response()
            ->view('sitemap', compact('posts'))
            ->header('Content-Type', 'application/xml');
    }
}Code language: PHP (php)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  @foreach($posts as $post)
    <url>
      <loc>{{ url('/posts/' . $post->slug) }}</loc>
      <lastmod>{{ $post->updated_at->toAtomString() }}</lastmod>
    </url>
  @endforeach
</urlset>Code language: PHP (php)
```

The XML sitemap lists all post URLs. Submit it to Google Search Console for faster indexing.

Wrapping Up

Optimizing Laravel for SEO requires three key pieces: unique slugs, dynamic meta tags, and an XML sitemap. With the optimized duplicate slug solution and database unique index, you avoid conflicts. With meta tags and Open Graph, you improve visibility and click-throughs. With sitemaps, you make crawling efficient. Together, these ensure Laravel apps are SEO-ready out of the box.

What’s Next

Keep building your SEO toolkit with these guides:

- How to Generate SEO-Friendly URLs and Slugs in Laravel
- How to Build an XML Sitemap Generator in Laravel
- Adding Meta Tags and Open Graph Data Dynamically in Laravel