1v0 Ship v1.0 Faster

10 Proven Ways to Optimize Laravel for High Traffic

10 Proven Ways to Optimize Laravel for High Traffic

When your Laravel app starts attracting high traffic, performance bottlenecks become critical. A slow site means lost users and revenue. Laravel provides multiple tools, and with the right best practices, you can scale your app to handle thousands of requests per second. In this guide, we'll cover 10 proven optimization techniques—from caching and queues to database tuning and server setup—plus link you to in-depth tutorials for deeper dives.

1 – Enable Config, Route & View Caching

Laravel allows caching of configuration, routes, and compiled views to reduce overhead.

```
php artisan config:cache
php artisan route:cache
php artisan view:cacheCode language: Bash (bash)
```

These commands compile files into optimized PHP arrays, drastically cutting down load times. Run them in your deployment pipeline.

2 – Use Query Caching with Redis

Database queries often become a bottleneck. Cache results in Redis to reduce repeated calls.

```
$posts = Cache::remember('latest_posts', 60, function () {
    return Post::latest()->take(20)->get();
});Code language: PHP (php)
```

This caches the query for 60 seconds. Subsequent requests read from Redis instead of hitting the DB. For a detailed comparison of cache stores, see Caching Strategies in Laravel: Redis vs Database vs File.

3 – Optimize Database with Indexes

Adding indexes to frequently queried columns can speed up lookups dramatically.

```
// database/migrations/add_index_to_users_email.php
Schema::table('users', function (Blueprint $table) {
    $table->index('email');
});Code language: PHP (php)
```

This adds an index to the `email` column. Always analyze queries with `EXPLAIN` in MySQL/Postgres to confirm. For a full guide, check How to Speed Up Laravel with Database Indexing.

4 – Reduce N+1 Queries with Eager Loading

The N+1 query problem slows down high-traffic apps. Use `with()` to fetch relationships in fewer queries.

```
$users = User::with('posts.comments')->get();Code language: PHP (php)
```

This loads users, their posts, and comments in one go. For best practices, read Eager Loading vs Lazy Loading in Laravel: Best Practices.

5 – Use Queues for Heavy Jobs

Don’t let emails, reports, or API calls block requests. Offload them to queues.

```
// Dispatching a queued job
SendWelcomeEmail::dispatch($user);Code language: PHP (php)
```

This pushes work into your queue system (Redis, Beanstalkd, SQS). Your app responds instantly while workers process jobs in the background. Learn more in How to Use Laravel Queues for Faster Performance.

6 – Optimize Asset Delivery

Large CSS/JS files slow requests. Use Laravel Mix or Vite to minify and version assets, and serve them via CDN.

```
npm run buildCode language: Bash (bash)
```

This produces minified, cache-busted files. Use `mix()` or `vite()` in Blade to reference the correct versions.

7 – Scale with Octane

Laravel Octane runs on Swoole or RoadRunner, keeping the app in memory between requests for lightning-fast responses.

```
composer require laravel/octane
php artisan octane:startCode language: Bash (bash)
```

Octane removes PHP's per-request bootstrapping. For advanced scaling, see Optimizing Laravel for High Concurrency with Octane.

8 – Monitor Performance with Telescope

Laravel Telescope gives deep insight into queries, requests, jobs, and cache hits. Perfect for diagnosing bottlenecks in real time.

```
composer require laravel/telescope --dev
php artisan telescope:install
php artisan migrateCode language: Bash (bash)
```

Once installed, visit `/telescope` to monitor app activity. See our detailed guide Using Laravel Telescope to Debug Performance Issues.

9 – Use PHP OPcache

Enable OPcache in your PHP setup. It caches compiled bytecode, cutting response times in half.

```
; php.ini
opcache.enable=1
opcache.memory_consumption=128
opcache.max_accelerated_files=10000Code language: TOML, also INI (ini)
```

This ensures your app's PHP code is compiled once and reused across requests, reducing CPU load.

10 – Horizontal Scaling & Load Balancing

For very high traffic, scale horizontally. Run multiple app servers behind a load balancer, and use a shared cache/database layer.

```
upstream laravel_app {
    server app1:9000;
    server app2:9000;
}

server {
    location / {
        proxy_pass http://laravel_app;
    }
```

```
}Code language: Nginx (nginx)
```

This Nginx config balances requests across two Laravel app servers. Combine with a managed DB cluster and Redis cache for best results.

Wrapping Up

We explored 10 proven techniques to optimize Laravel for high traffic: caching, indexing, eager loading, queues, assets, Octane, monitoring, OPcache, and load balancing. Combined, these approaches ensure your app scales smoothly. Start with quick wins (config cache, query indexes) and move towards advanced scaling (Octane, horizontal scaling) as traffic grows.

What's Next

- How to Use Laravel Queues for Faster Performance — dive deeper into offloading heavy jobs.
- Caching Strategies in Laravel: Redis vs Database vs File — detailed cache storage comparisons.
- Using Laravel Telescope to Debug Performance Issues — learn to monitor queries, jobs, and bottlenecks.