Supercharge Your Laravel Apps with the Concurrency Facade

Laravel 12 introduces a game-changing feature that can dramatically improve your application's performance: the **Concurrency facade**. If you've ever found yourself waiting for multiple slow operations to complete sequentially, this feature is about to become your new best friend.

The Problem: Sequential Execution Bottleneck

Picture this scenario: you're building a dashboard that needs to fetch data from multiple sources:

- User count from the database
- Today's order statistics
- Revenue calculations
- External API calls for weather data
- Third-party analytics

Traditionally, these operations run one after another. If each takes 500ms, you're looking at 2.5 seconds of waiting time. That's a terrible user experience.

```
// The old way - sequential execution
$users = DB::table('users')->count(); // 500ms
$orders = DB::table('orders')->count(); // 500ms
$revenue = Order::sum('total'); // 500ms
$weather = Http::get('api.weather.com')->json(); // 500ms
$analytics = Http::get('api.analytics.com')->json(); // 500ms

// Total time: ~2.5 seconds
```

The Solution: Laravel Concurrency

Laravel's Concurrency facade allows you to execute independent tasks **simultaneously** in separate PHP processes. Those same five operations? They now complete in roughly 500ms total.

```
use Illuminate\Support\Facades\Concurrency;

// The new way - concurrent execution
[$users, $orders, $revenue, $weather, $analytics] = Concurrency::run([
    fn () => DB::table('users')->count(),
    fn () => DB::table('orders')->count(),
    fn () => Order::sum('total'),
    fn () => Http::get('api.weather.com')->json(),
    fn () => Http::get('api.analytics.com')->json(),
]);
```

```
// Total time: ~500ms
```

That's a **5x performance improvement** with just a few lines of code!

How It Works Under the Hood

Laravel's implementation is elegant:

1. **Serialization**: Your closures are serialized
2. **Process Spawning**: Each closure is sent to a separate PHP process via hidden Artisan commands
3. **Parallel Execution**: All processes run simultaneously
4. **Result Collection**: Results are serialized back to the parent process
5. **Return**: You get an array of results in the same order

```
// Behind the scenes, Laravel does something like this:
// Process 1: php artisan laravel:run-closure [serialized-closure-1]
// Process 2: php artisan laravel:run-closure [serialized-closure-2]
// Process 3: php artisan laravel:run-closure [serialized-closure-3]
// All running at the same time!
```

Three Drivers for Different Needs

Laravel provides three concurrency drivers:

1. Process Driver (Default)

The standard driver that works everywhere. It spawns separate PHP processes for each task.

```
// Uses process driver by default
$results = Concurrency::run([
    fn () => $this->heavyTask1(),
    fn () => $this->heavyTask2(),
]);
```

2. Fork Driver (Fastest)

Offers better performance by forking the current process. **Only works in CLI context** (Artisan commands, queue workers), not during web requests.

```
# First, install the required package
composer require spatie/fork
```

```
// Use fork driver for CLI commands
$results = Concurrency::driver('fork')->run([
    fn () => $this->processLargeFile(),
    fn () => $this->generateReport(),
    fn () => $this->calculateStatistics(),
]);
```

3. Sync Driver (Testing)

Executes closures sequentially without any concurrency. Perfect for testing environments.

```
// In your tests
$results = Concurrency::driver('sync')->run([
    fn () => $this->task1(),
    fn () => $this->task2(),
]);
```

Real-World Use Cases

Let me share some practical examples from real projects.

Dashboard Data Loading

```
public function getDashboardData()
{
    [$totalUsers, $activeUsers, $todayOrders, $revenue, $topProducts] =
        Concurrency::run([
            fn () => User::count(),
            fn () => User::where('last_login', '>', now()->subDays(7))->count(),
            fn () => Order::whereDate('created_at', today())->count(),
            fn () => Order::whereDate('created_at', today())->sum('total'),
            fn () => Product::withCount('orders')
                ->orderBy('orders_count', 'desc')
                ->limit(10)
                ->get(),
        ]);

    return compact('totalUsers', 'activeUsers', 'todayOrders', 'revenue', 'topProducts');
}
```

Multiple API Integrations

This is particularly useful if you're working with multiple suppliers or external services:

```
use Illuminate\Support\Facades\Concurrency;

public function aggregateTireInventory()
```

```
{
    // Fetch from multiple tire suppliers simultaneously
    [$interSprint, $tomket, $vandenBan] = Concurrency::run([
        fn () => $this->fetchInterSprintCatalog(),
        fn () => $this->fetchTomketProducts(),
        fn () => $this->fetchVandenBanInventory(),
    ]);

    // Merge and normalize data
    return $this->normalizeProducts(
        collect($interSprint)
            ->merge($tomket)
            ->merge($vandenBan)
    );
}

private function fetchInterSprintCatalog()
{
    return Http::withToken(config('services.intersprint.token'))
        ->get('https://api.intersprint.com/products')
        ->json();
}
```

Image Processing Pipeline

```
public function processProductImages(string $imagePath)
{
    [$thumbnail, $medium, $large, $optimized] = Concurrency::run([
        fn () => $this->generateThumbnail($imagePath),
        fn () => $this->generateMediumSize($imagePath),
        fn () => $this->generateLargeSize($imagePath),
        fn () => $this->optimizeOriginal($imagePath),
    ]);

    return [
        'thumbnail' => $thumbnail,
        'medium' => $medium,
        'large' => $large,
        'original' => $optimized,
    ];
}
```

Batch Data Processing

```
public function generateMonthlyReports()
{
    $months = collect(['January', 'February', 'March', 'April']);

    $reports = Concurrency::run(
        $months->map(fn ($month) =>
            fn () => $this->generateReport($month)
```

```
        )->all()
    );

    return collect($reports)
        ->zip($months)
        ->mapWithKeys(fn ($item) => [$item[1] => $item[0]]);
}
```

Deferred Execution: Fire and Forget

Sometimes you don't need the results immediately. Use `defer()` to execute tasks **after the HTTP response is sent**:

```
use Illuminate\Support\Facades\Concurrency;

public function handleUserAction(Request $request)
{
    // Process the main request
    $result = $this->processUserAction($request);

    // These run after response is sent
    Concurrency::defer([
        fn () => Analytics::track('user_action', $request->all()),
        fn () => $this->sendSlackNotification($result),
        fn () => Cache::tags('stats')->flush(),
        fn () => $this->updateSearchIndex($result),
    ]);

    return response()->json($result);
}
```

This is perfect for:

- Analytics tracking
- Cache invalidation
- Logging
- Notifications
- Search index updates

Configuration and Best Practices

Change the Default Driver

```
php artisan config:publish concurrency
```

```
// config/concurrency.php
return [
    'default' => env('CONCURRENCY_DRIVER', 'process'),

    'drivers' => [
        'process' => [],
        'fork' => [],
        'sync' => [],
    ],
];
```

Environment-Specific Drivers

```
# .env
CONCURRENCY_DRIVER=process

# .env.testing
CONCURRENCY_DRIVER=sync

# .env.local (for CLI development)
CONCURRENCY_DRIVER=fork
```

Best Practices

DO:

- Use for independent tasks that don't depend on each other
- Use for I/O-bound operations (API calls, database queries)
- Use `defer()` for non-critical background tasks
- Test with the `sync` driver in your test suite

DON'T:

- Use for tasks that share state or depend on execution order
- Use for CPU-intensive tasks in web requests (consider queues instead)
- Use the `fork` driver during HTTP requests
- Forget to handle exceptions in concurrent closures

Error Handling

```
use Illuminate\Support\Facades\Concurrency;

try {
    [$result1, $result2] = Concurrency::run([
        fn () => $this->mayFailTask1(),
        fn () => $this->mayFailTask2(),
    ]);
```

```
} catch (\Throwable $e) {
    // Handle failures
    Log::error('Concurrent task failed', [
        'message' => $e->getMessage(),
        'trace' => $e->getTraceAsString(),
    ]);

    // Fallback logic
    $result1 = $this->getDefaultValue1();
    $result2 = $this->getDefaultValue2();
}
```

Performance Benchmarks

Here's a real-world comparison from a dashboard I optimized:

Approach	Execution Time	Improvement
Sequential	3.2s	Baseline
Concurrent (process)	0.8s	**4x faster**
Concurrent (fork)	0.6s	**5.3x faster**

The difference becomes even more dramatic as you add more concurrent tasks.

When NOT to Use Concurrency

Concurrency isn't always the answer. Consider alternatives in these cases:

- **Dependent tasks**: If task B needs task A's result, run them sequentially
- **Heavy CPU tasks**: Use Laravel Queues with multiple workers instead
- **Database transactions**: Keep transactional operations sequential
- **Rate-limited APIs**: You might hit rate limits faster with concurrent requests

Combining with Other Laravel Features

With Cache

```
[$users, $orders] = Concurrency::run([
    fn () => Cache::remember('users.count', 3600,
        fn () => User::count()
    ),
    fn () => Cache::remember('orders.today', 3600,
        fn () => Order::whereDate('created_at', today())->count()
    ),
]);
```

With Queues

```
// Dispatch multiple queue jobs concurrently
Concurrency::defer([
    fn () => ProcessOrderJob::dispatch($order1),
    fn () => ProcessOrderJob::dispatch($order2),
    fn () => ProcessOrderJob::dispatch($order3),
]);
```

With Events

```
// Trigger multiple events after response
Concurrency::defer([
    fn () => event(new OrderPlaced($order)),
    fn () => event(new InventoryUpdated($product)),
    fn () => event(new CustomerNotified($customer)),
]);
```

Conclusion

Laravel's Concurrency facade is a powerful tool that can dramatically improve your application's performance with minimal code changes. By executing independent tasks simultaneously, you can reduce response times, improve user experience, and make better use of your server resources.

The best part? It's incredibly easy to implement. Start by identifying slow, independent operations in your application and wrap them in `Concurrency::run()`. You'll be amazed at the results.

Quick Reference

```
// Basic usage
[$a, $b] = Concurrency::run([
    fn () => $taskA(),
    fn () => $taskB(),
]);

// Custom driver
$results = Concurrency::driver('fork')->run([...]);

// Deferred execution
Concurrency::defer([
    fn () => $task1(),
    fn () => $task2(),
]);

// Change default driver
```

```
// config/concurrency.php
'default' => 'process',
```

Got questions or want to share your concurrency success stories? Drop a comment below or reach out on Twitter @cherradiX.

Happy coding!