How to Schedule Jobs in Laravel with Task Scheduling

Laravel’s task scheduling system allows you to automate repetitive jobs such as clearing caches, sending out reports, or syncing data. Instead of creating multiple cron jobs on the server, you define all of your scheduled tasks in code and keep them version controlled. This makes it easier to manage and deploy automation tasks consistently.

Setting Up the Scheduler

```bash
* * * * * php /path-to-your-app/artisan schedule:run >> /dev/null 2>&1
```
Code language: Bash (bash)

You only need a single cron entry that runs Laravel’s scheduler every minute. From there, Laravel decides which tasks need to execute. This approach keeps your automation clean and centralized inside your application.

Defining Scheduled Tasks

```php
// app/Console/Kernel.php

protected function schedule(Schedule $schedule)
{
    $schedule->command('emails:send')->dailyAt('09:00');
    $schedule->job(new CleanUpTempFiles)->hourly();
}
```
Code language: PHP (php)

You define tasks inside `app/Console/Kernel.php` using the `schedule` method. You can schedule Artisan commands, queued jobs, or closures. Laravel provides methods like `dailyAt()`, `hourly()`, and `weeklyOn()` to customize when tasks run.

Different Frequency Options

```php
$schedule->command('cache:clear')->everyMinute();
$schedule->command('backup:run')->twiceDaily(1, 13);
$schedule->command('reports:send')->weeklyOn(1, '08:00'); // Monday 8am
$schedule->command('cleanup:archive')->monthlyOn(1, '02:00');
```
Code language: PHP (php)

Laravel includes expressive frequency methods to fit your use case. You can schedule tasks every minute, twice daily, weekly on a specific day, or monthly at a given time.

Environment-Based Scheduling

```php
$schedule->command('emails:send')
    ->daily()
    ->environments(['production']);
```
Code language: PHP (php)

You can restrict tasks to specific environments. For example, you may want to send emails only in production and skip them in your local or staging environments.

Output Logging

```
$schedule->command('reports:generate')
    ->daily()
    ->appendOutputTo(storage_path('logs/schedule.log'));
```
Code language: PHP (php)

When running scheduled commands, capturing their output helps with debugging. Use `appendOutputTo()` or `sendOutputTo()` to direct the output to log files.

Practical Maintenance Tasks

```
$schedule->command('model:prune')->daily();
$schedule->command('cache:clear')->weekly();
$schedule->command('db:backup')->dailyAt('01:00');
```
Code language: PHP (php)

Typical tasks include pruning old models, clearing caches, or running database backups. Automating these jobs improves reliability and reduces manual overhead.

Using Closures in Scheduling

```php
$schedule->call(function () {
    \Log::info('Scheduled closure executed at ' . now());
})->everyFiveMinutes();
```
Code language: PHP (php)

Closures let you define quick tasks inline without creating full commands. This is useful for logging, temporary cleanups, or testing scheduled execution.

Running Scheduled Events in the Background

```php
$schedule->command('reports:generate')
    ->dailyAt('23:00')
    ->withoutOverlapping()
    ->runInBackground();
```
Code language: PHP (php)

`withoutOverlapping()` ensures that a job won't run again if the previous instance hasn't finished. Adding `runInBackground()` allows long-running tasks to execute without blocking others.

Building a UI to Track Completed Jobs

Sometimes you need visibility into which scheduled tasks or queued jobs have already run. While Laravel Horizon provides a great dashboard, you can also build a lightweight UI yourself for audit and monitoring purposes.

Migration: Job Runs Table

```
// database/migrations/2025_09_02_000001_create_job_runs_table.php
Schema::create('job_runs', function (Blueprint $table) {
    $table->id();
    $table->string('job_name');
    $table->string('queue')->nullable();
    $table->timestamp('started_at')->nullable();
    $table->timestamp('finished_at')->nullable();
    $table->unsignedInteger('duration_ms')->nullable();
    $table->string('status'); // success | failed
    $table->longText('exception')->nullable();
    $table->timestamps();
});Code language: PHP (php)
```

This table logs each executed job with status, duration, and exception details if applicable.

Queue Events

```
// app/Providers/EventServiceProvider.php
use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobFailed;

public function boot(): void
{
    Queue::after(function (JobProcessed $event) {
        \App\Models\JobRun::create([
            'job_name' => $event->job->resolveName(),
            'status' => 'success',
            'started_at' => now()->subSeconds(1),
            'finished_at' => now(),
        ]);
    });

    Queue::failing(function (JobFailed $event) {
        \App\Models\JobRun::create([
            'job_name' => $event->job->resolveName(),
            'status' => 'failed',
            'exception' => $event->exception->getMessage(),
        ]);
    });
}
```
Code language: PHP (php)

Here we listen to job lifecycle events and insert records into the `job_runs` table. You can enrich it with duration, queue name, or custom tags.

Blade UI Example

```
// resources/views/job-runs/index.blade.php
<table class="table-auto w-full">
  <thead>
    <tr>
      <th>Job Name</th>
      <th>Status</th>
      <th>Started</th>
      <th>Finished</th>
      <th>Error</th>
    </tr>
  </thead>
  <tbody>
    @foreach($runs as $run)
    <tr>
      <td>{{ $run->job_name }}</td>
      <td>{{ $run->status }}</td>
      <td>{{ $run->started_at }}</td>
      <td>{{ $run->finished_at }}</td>
      <td>{{ $run->exception }}</td>
    </tr>
    @endforeach
  </tbody>
</table>
```
Code language: PHP (php)

This Blade view lists job history, making it easy to monitor success/failure patterns. You can add pagination, filtering, or search for larger datasets.

Wrapping Up

Laravel's scheduler streamlines automation by letting you manage all recurring jobs inside your codebase. With support for different frequencies, environment restrictions, logging, and background execution, you can cover almost any use case. For visibility, building a simple UI to list completed jobs ensures transparency and helps with debugging and auditing.

What's Next

To keep building robust background processes and automation, explore these articles:

- Using Laravel Telescope to Debug Performance Issues
- Laravel Horizon vs Queue Workers: Which One Should You Use?
- How to Use Laravel Queues for Faster Performance