Task Scheduling

#Introduction

In the past, you may have generated a Cron entry for each task you needed to schedule on your server. However, this can quickly become a pain, because your task schedule is no longer in source control and you must SSH into your server to add additional Cron entries.

Laravel's command scheduler allows you to fluently and expressively define your command schedule within Laravel itself. When using the scheduler, only a single Cron entry is needed on your server. Your task schedule is defined in the `app/Console/Kernel.php` file's `schedule` method. To help you get started, a simple example is defined within the method.

Starting The Scheduler

When using the scheduler, you only need to add the following Cron entry to your server. If you do not know how to add Cron entries to your server, consider using a service such as Laravel Forge which can manage

the Cron entries for you:

```
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

This Cron will call the Laravel command scheduler every minute. When the `schedule:run` command is executed, Laravel will evaluate your scheduled tasks and runs the tasks that are due.

#Defining Schedules

You may define all of your scheduled tasks in the `schedule` method of the `App\Console\Kernel` class. To get started, let's look at an example of scheduling a task. In this example, we will schedule a `Closure` to be called every day at midnight. Within the `Closure` we will execute a database query to clear a table:

```
<?php

namespace App\Console;

use DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        //
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
```

```
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}
```

In addition to scheduling using Closures, you may also using invokable objects. Invokable objects are simple PHP classes that contain an `__invoke` method:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

Scheduling Artisan Commands

In addition to scheduling Closure calls, you may also schedule Artisan commands and operating system commands. For example, you may use the `command` method to schedule an Artisan command using either the command's name or class:

```
$schedule->command('emails:send --force')->daily();


$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

Scheduling Queued Jobs

The `job` method may be used to schedule a queued job. This method provides a convenient way to schedule jobs without using the `call` method to manually create Closures to queue the job:

```
$schedule->job(new Heartbeat)->everyFiveMinutes();


// Dispatch the job to the "heartbeats" queue...
$schedule->job(new Heartbeat, 'heartbeats')->everyFiveMinutes();
```

Scheduling Shell Commands

The `exec` method may be used to issue a command to the operating system:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Schedule Frequency Options

Of course, there are a variety of schedules you may assign to your task:

Method	Description
`->cron('* * * * *');`	Run the task on a custom Cron schedule
`->everyMinute();`	Run the task every minute
`->everyFiveMinutes();`	Run the task every five minutes
`->everyTenMinutes();`	Run the task every ten minutes
`->everyFifteenMinutes();`	Run the task every fifteen minutes
`->everyThirtyMinutes();`	Run the task every thirty minutes
`->hourly();`	Run the task every hour
`->hourlyAt(17);`	Run the task every hour at 17 mins past the hour
`->daily();`	Run the task every day at midnight
`->dailyAt('13:00');`	Run the task every day at 13:00
`->twiceDaily(1, 13);`	Run the task daily at 1:00 & 13:00
`->weekly();`	Run the task every week
`->weeklyOn(1, '8:00');`	Run the task every week on Monday at 8:00
`->monthly();`	Run the task every month
`->monthlyOn(4, '15:00');`	Run the task every month on the 4th at 15:00
`->quarterly();`	Run the task every quarter

Method	Description
`->yearly();`	Run the task every year
`->timezone('America/New_York');`	Set the timezone

These methods may be combined with additional constraints to create even more finely tuned schedules that only run on certain days of the week. For example, to schedule a command to run weekly on Monday:

```
// Run once per week on Monday at 1 PM...
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')
          ->weekdays()
          ->hourly()
          ->timezone('America/Chicago')
          ->between('8:00', '17:00');
```

Below is a list of the additional schedule constraints:

Method	Description
`->weekdays();`	Limit the task to weekdays
`->sundays();`	Limit the task to Sunday
`->mondays();`	Limit the task to Monday
`->tuesdays();`	Limit the task to Tuesday
`->wednesdays();`	Limit the task to Wednesday
`->thursdays();`	Limit the task to Thursday
`->fridays();`	Limit the task to Friday
`->saturdays();`	Limit the task to Saturday

Method	Description
`->between($start, $end);`	Limit the task to run between start and end times
`->when(Closure);`	Limit the task based on a truth test

Between Time Constraints

The `between` method may be used to limit the execution of a task based on the time of day:

```
$schedule->command('reminders:send')
                     ->hourly()
                     ->between('7:00', '22:00');
```

Similarly, the `unlessBetween` method can be used to exclude the execution of a task for a period of time:

```
$schedule->command('reminders:send')
                     ->hourly()
                     ->unlessBetween('23:00', '4:00');
```

Truth Test Constraints

The `when` method may be used to limit the execution of a task based on the result of a given truth test. In other words, if the given `Closure` returns `true`, the task will execute as long as no other constraining conditions prevent the task from running:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

The `skip` method may be seen as the inverse of `when`. If the `skip` method returns `true`, the scheduled task will not be executed:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

When using chained `when` methods, the scheduled command will only execute if all `when` conditions return `true`.

Timezones

Using the `timezone` method, you may specify that a scheduled task's time should be interpreted within a given timezone:

```
$schedule->command('report:generate')
         ->timezone('America/New_York')
         ->at('02:00')
```

> Remember that some timezones utilize daylight savings time. When daylight saving time changes occur, your scheduled task may run twice or even not run at all. For this reason, we recommend avoiding timezone scheduling when possible.

Preventing Task Overlaps

By default, scheduled tasks will be run even if the previous instance of the task is still running. To prevent this, you may use the `withoutOverlapping` method:

```
$schedule->command('emails:send')->withoutOverlapping();
```

In this example, the `emails:send` Artisan command will be run every minute if it is not already running. The `withoutOverlapping` method is especially useful if you have tasks that vary drastically in their execution time, preventing you from predicting exactly how long a given task will take.

If needed, you may specify how many minutes must pass before the "without overlapping" lock expires. By default, the lock will expire after 24 hours:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

Running Tasks On One Server

> To utilize this feature, your application must be using the `memcached` or `redis` cache driver as your application's default cache driver. In addition, all servers must be communicating with the

same central cache server.

If your application is running on multiple servers, you may limit a scheduled job to only execute on a single server. For instance, assume you have a scheduled task that generates a new report every Friday night. If the task scheduler is running on three worker servers, the scheduled task will run on all three servers and generate the report three times. Not good!

To indicate that the task should run on only one server, use the `onOneServer` method when defining the scheduled task. The first server to obtain the task will secure an atomic lock on the job to prevent other servers from running the same task at the same time:

```
$schedule->command('report:generate')
                 ->fridays()
                 ->at('17:00')
                 ->onOneServer();
```

Maintenance Mode

Laravel's scheduled tasks will not run when Laravel is in maintenance mode, since we don't want your tasks to interfere with any unfinished maintenance you may be performing on your server. However, if you would like to force a task to run even in maintenance mode, you may use the `evenInMaintenanceMode` method:

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

Task Output

The Laravel scheduler provides several convenient methods for working with the output generated by scheduled tasks. First, using the `sendOutputTo` method, you may send the output to a file for later inspection:

```
$schedule->command('emails:send')
          ->daily()
          ->sendOutputTo($filePath);
```

If you would like to append the output to a given file, you may use the `appendOutputTo` method:

```
$schedule->command('emails:send')
         ->daily()
         ->appendOutputTo($filePath);
```

Using the `emailOutputTo` method, you may e-mail the output to an e-mail address of your choice. Before e-mailing the output of a task, you should configure Laravel's e-mail services:

```
$schedule->command('foo')
         ->daily()
         ->sendOutputTo($filePath)
         ->emailOutputTo('foo@example.com');
```

> The `emailOutputTo`, `sendOutputTo` and `appendOutputTo` methods are exclusive to the `command` and `exec` methods.

#Task Hooks

Using the `before` and `after` methods, you may specify code to be executed before and after the scheduled task is complete:

```
$schedule->command('emails:send')
         ->daily()
         ->before(function () {
             // Task is about to start...
         })
         ->after(function () {
             // Task is complete...
         });
```

Pinging URLs

Using the `pingBefore` and `thenPing` methods, the scheduler can automatically ping a given URL before or after a task is complete. This method is useful for notifying an external service, such as Laravel Envoyer, that your scheduled task is commencing or has finished execution:

```
$schedule->command('emails:send')
         ->daily()
         ->pingBefore($url)
         ->thenPing($url);
```

Using either the `pingBefore($url)` or `thenPing($url)` feature requires the Guzzle HTTP library. You can add Guzzle to your project using the Composer package manager:

```
composer require guzzlehttp/guzzle
```