Back to Scaling PHP Applications guides

A Complete Guide to Task Scheduling in Laravel

PHP Monitoring Job Scheduling Laravel

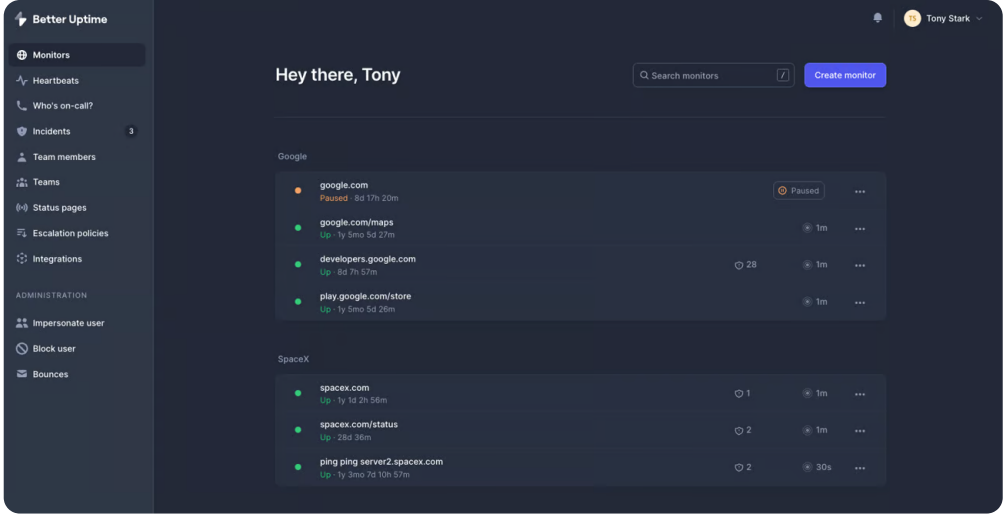
Eric Hu

UPDATED ON MARCH 17, 2023

Task scheduling is a useful technique for automating various repetitive tasks based on a schedule. Such tasks or jobs may be mission critical in nature (such as backing up a database), or it may be as simple as send a weekly email to yourself or your customers. Manually running such jobs can get tedious real quick so a better solution is to automate and monitor them so they can run predictably in a timely fashion.

The traditional way to schedule tasks on applications deployed to Linux servers is through the cron utility. However, such system must be implemented separately from the application which can be quite limiting.

Laravel offers a holistic approach to task scheduling through its command scheduler which allows you to schedule tasks within the application itself. In this tutorial, we will explore how to create scheduled jobs in Laravel, and we will also implement a monitoring solution to help you to promptly notify you if a scheduled task fails or doesn't run as expected.



Prerequisites

Before proceeding with this article, ensure that you have access to a Linux machine with recent versions of Cron, PHP, and Composer installed.

You should also create a new Laravel project so that you may test the code snippets in this tutorial. Assuming PHP and Composer are installed, you can proceed to create a Laravel project using the following command:

```
$ composer create-project laravel/laravel <project_name>
```

Getting started

Your new Laravel project should have the following structure:

```
.
├── README.md
```

```
├── app
│   ├── Console
│   │   └── Kernel.php
│   ├── Exceptions
│   │   └── Handler.php
│   ├── Http
│   │   ├── Controllers
│   │   │   ├── Controller.php
│   │   ├── Kernel.php
│   │   └── Middleware
│   ├── Models
│   └── Providers
├── artisan
├── bootstrap
├── composer.json
├── composer.lock
├── config
├── database
├── package.json
├── php_errors.log
├── phpunit.xml
├── public
├── resources
├── routes
├── storage
├── tests
└── vite.config.js
```

For this tutorial, we only care about the `Console` directory

`app\Console\Kernel.php` file.

Go ahead and open the file with the following command:

```
$ nano app\Console\Kernel.php
```

app\Console\Kernel.php

```
<?php

namespace App\Console;

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

```
class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $sche
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        // $schedule->command("inspire")->hourly();
    }

    /**
     * Register the commands for the application.
     *
     * @return void
     */
    protected function commands()
    {
        $this->load(__DIR__ . "/Commands");

        require base_path("routes/console.php");
    }
}
```

Notice the highlighted `schedule()` function above. It's where you can schedule your tasks, and Laravel provides an example of how it is done within the function:

```
$schedule->command("inspire")->hourly();
```

This line is commented out by default, but you should uncomment it before going forward. The `$schedule` variable is an instance of the `Schedule` class built into Laravel. Its `command()` method will schedule an Artisan command which displays an inspiring quote in the console.

To view a list of all available Artisan commands, run the command below in the terminal from your project root:

```
$ php artisan list
```

You will observe the command list alongside a brief description of what they do:

Output

```
Laravel Framework 9.33.0

. . .

Available commands:
 about Display basic information about your application
 clear-compiled Remove the compiled class file
 . . .
 view
 view:cache Compile all of the application's Blade templates
 view:clear Clear all compiled view files
```

After choosing a command, you must specify how often you wish to execute it. In the default example, the `hourly()` method executes the `inspire` command every hour. For demonstration purposes, change it to `everyMinute()` so you can see the effect of the task scheduling much quicker.

```
$schedule->command('inspire')->everyMinute();
```

After making the change, invoke the scheduler by executing the command below:

```
$ php artisan schedule:run
```

The `schedule:run` command will go through all scheduled tasks and determine if Laravel should execute the task based

on the current time. For example, `hourly()` will schedule a task to be executed on the hour mark (xx:00) while `everyFiveMinutes()` executes the job when the current time is xx:x0, xx:x5, xx:10, etc. If no task is currently scheduled to be executed, the following output will be printed to the terminal:

```
INFO  No scheduled commands are ready to run.
```

However, since we are using `everyMinute()` in our example, you should observe that the task is executed and the following output is printed to the terminal:

Output

```
 2022-10-06 21:42:39 Running ['artisan' inspire]
...................2,878ms DONE
 ↳ '/usr/bin/php8.1' 'artisan' inspire > '/dev/null' 2>&1
```

Notice that a log message describing the executed command is printed, but the result isn't displayed. That's because Laravel automatically ignores the output of a scheduled task by forwarding it to `/dev/null` (see second line of above output) since they are typically run in the background.

If you need to utilize the output of your scheduled task, you can place it in a file for later inspection through the `sendOutputTo()` method:

```
$schedule
    ->command("inspire")
    ->everyMinute()
    ->sendOutputTo("scheduler-output.log");
```

After making the above change, invoke the scheduler again and view the contents of the `scheduler-output.log` file that is subsequently created:

```
$ cat scheduler-output.log
```

Output

```
 " Knowing is not enough; we must apply. Being willing is not enough; we must do. "
 — Leonardo da Vinci
```

Note that the `sendOutputTo()` method overwrites the contents of its file argument for each invocation. If this is not desired, you can use the `appendOutputTo()` method that appends the command output to the end of the file instead.

```
$schedule
    ->command("inspire")
    ->everyMinute()
    ->appendOutputTo("scheduler-output.log");
```

Run the scheduler a few times, and observe that the file is no longer being overwritten:

```
$ cat scheduler-output.log
```

```
 " Simplicity is the consequence of refined emotions. "
 — Jean D'Alembert
```

```
“ Order your soul. Reduce your wants. ”
— Augustine


“ Simplicity is the essence of happiness. ”
— Cedric Bledsoe


“ I begin to speak only when I am certain what I will say i
— Cato the Younger
```

A problem with the `schedule:run` command is that it only invokes the scheduler once which means that the scheduled tasks are executed just once to ensure tasks are run as scheduled, the scheduler needs to be constantly running and this is achieved by using the `schedule:work` command instead as shown below:

```
$ php artisan schedule:work
```

This command will run in the foreground and invoke the `schedule:run` command every minute until it is terminated by pressing `Ctrl-C`. It is useful for testing your code during development so you don't have to manually invoke the scheduler each time.

In a production environment, you should execute the scheduler in the background through a Cron Job as shown below:

```
$ crontab -e
```

```
```

```
it this file to introduce tasks to be run by cron.
.
* * * cd <path_to_your_project> && php artisan schedule:run >> /
```


Scheduling system commands

Aside from Artisan commands, you can also schedule the execution of any system command using the `exec()` method as follows:

```
$schedule->exec("bash script.sh")->hourly();
```

The `sendOutputTo()` and `appendOutputTo()` methods discussed earlier can also be used to capture the output of system commands.

Scheduling functions

If you need to schedule a function in Laravel, you can use the `call()` method and pass in a closure that wraps the function as follows:

```
<?php

namespace App\Console;

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

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $sche
```

```
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule
            ->call(function () {
                DB::table("users")
                    ->where("status", "inactive")
                    ->delete();
            })
            ->daily();
    }
}
```

The scheduled job will run daily in this example and delete all inactive users.

In addition to scheduling closures, you may also schedule invokable objects with the `call()` method. These are PHP classes that contain an `__invoke()` method. When a new instance of the class is fed to the `call()` method, its `__invoke()` method will be executed according to the specified schedule.

```
$schedule->call(new SomeClass)->hourly();
```

Scheduling queued jobs

Queued jobs are typically long-running tasks that are processed in the background to avoid interfering with the application's main processes. Such tasks can be executed on a defined schedule by using the `job()` method on the scheduler.

For instance, you can use this feature to update the search index of your application once every hour as follows:

```
use App\Jobs\UpdateSearchIndex;

$schedule->job(new UpdateSearchIndex)->hourly();
```

If you are using a third-party service (such as Amazon SQS) to handle your queued jobs, you also need to specify the name of the job and the service you are using so that the appropriate queue connection will be used to queue the job:

```
$schedule->job(new SomeQueuedJob, "myJob", "sqs")->hourly();
```

Listing all scheduled tasks

You can obtain a list of all scheduled tasks using the command below. It displays each scheduled task and the time of their next invocation:

```
$ php artisan schedule:list
```

Output

```
0 * * * *  php artisan inspire
.............................. Next Due: 6 minutes from now
0 * * * *  bash script.sh
.....................................Next Due: 6 minutes
from now
0 0 * * *  Closure at: app/Console/Kernel.php:23
............Next Due: 4 hours from now
0 * * * *  App\Jobs\UpdateSearchIndex
.......................Next Due: 6 minutes from now
```

Running scheduled tasks in the background

When you have multiple tasks scheduled to run simultaneously, Laravel will execute them one by one according to how they are defined in the `app/Console/Kernel.php` file. However, if one task requires a long time to run, it will delay all subsequent tasks. Therefore, you might want to run your scheduled tasks in the background using the `runInBackground()` method, so that multiple jobs can run simultaneously.

```
$schedule
    ->exec("bash script.sh")
    ->hourly()
    ->runInBackground();
```

Note that `runInBackground()` can only be used for tasks scheduled with the `exec()` or `command()` methods only.

Setting the right schedule frequency

So far, we've only seen the `hourly()` and `everyMinute()` methods in action, but Laravel offers several other frequency options, which can be broadly classified into a few different categories which are discussed below.

Common frequency options

1. Minute options

```
$schedule->command("inspire")->everyMinute(); // Execute ever
$schedule->command("inspire")->everyTwoMinutes(); // Execute
$schedule->command("inspire")->everyThreeMinutes();
$schedule->command("inspire")->everyFourMinutes();
$schedule->command("inspire")->everyFiveMinutes();
$schedule->command("inspire")->everyTenMinutes();
$schedule->command("inspire")->everyFifteenMinutes();
$schedule->command("inspire")->everyThirtyMinutes();
```

2. Hourly options

```
$schedule->command("inspire")->hourly(); // Execute every hou
$schedule->command("inspire")->hourlyAt(15); // Execute every
$schedule->command("inspire")->everyOddHour();
$schedule->command("inspire")->everyTwoHours();
$schedule->command("inspire")->everyThreeHours();
$schedule->command("inspire")->everyFourHours();
$schedule->command("inspire")->everySixHours();
```

The hour-based options above will run their tasks at the first minute of the hour, but the `hourlyAt(n)` method can be used to run a task at the `nth` minute of the hour instead.

3. Daily options

```
$schedule->command("inspire")->daily(); // Execute daily at 0
$schedule->command("inspire")->dailyAt("10:15"); // Execute d
$schedule->command("inspire")->twiceDaily(10, 16); // Execute
$schedule->command("inspire")->twiceDailyAt(10, 16, 15); // E
```

4. Weekly options

```
$schedule->command("inspire")->weekly(); // Execute weekly on
$schedule->command("inspire")->weeklyOn(2, "8:00"); // Execut
```

The `weeklyOn()` method allows you to specify a day (Sunday: `0`, Monday: `1`, ... , Saturday: `6`) and a time, but you can also specify multiple days using an array:

```
$schedule->command("inspire")->weeklyOn([2, 4, 5], "8:00"); /
```

5. Monthly options

```
$schedule->command("inspire")->monthly(); // Execute on the f
$schedule->command("inspire")->monthlyOn(4, "15:00"); // Exec
$schedule->command("inspire")->twiceMonthly(1, 16, "13:00");
$schedule->command("inspire")->lastDayOfMonth("15:00"); // Ex
$schedule->command("inspire")->quarterly(); // Execute on the
```

6. Yearly options

```
$schedule->command("inspire")->yearly(); // Execute on the fi
$schedule->command("inspire")->yearlyOn(6, 1, "17:00"); // Ex
```

Constraint options

Constraint options are a special set of methods that defines additional constraints after you have specified the frequency. For example, you can first schedule a job to run weekly and then set constraints like this:

```
$schedule->command("inspire")->weekly()->weekdays(); // Runs
$schedule->command("inspire")->weekly()->weekends(); // Limit
$schedule->command("inspire")->weekly()->sundays();
$schedule->command("inspire")->weekly()->mondays();
$schedule->command("inspire")->weekly()->tuesdays();
$schedule->command("inspire")->weekly()->wednesdays();
$schedule->command("inspire")->weekly()->thursdays();
$schedule->command("inspire")->weekly()->fridays();
```

```
$schedule->command("inspire")->weekly()->saturdays();
$schedule->command("inspire")->weekly()->days([2, 5]); // Exe
```

The `days()` method takes an array of integers as input, which allows you to limit the task execution to specific days of the week.

Besides the day constraints, there are two additional time-based constraint methods, `between()` and `unlessBetween()`, which defines a range of times that a job is allowed to be executed.

```
// Execute hourly on weekdays, between 08:00 and 17:00
$schedule
    ->command("inspire")
    ->weekdays()
    ->hourly()
    ->between("8:00", "17:00");

// Execute hourly on weekdays, before 08:00 and after 17:00
$schedule
    ->command("inspire")
    ->weekdays()
    ->hourly()
    ->unlessBetween("8:00", "17:00");
```

A major benefit of using Laravel to schedule tasks is that you can use something other than the current time to trigger the execution of the tasks. For example, the `when()` method takes a callback function as its input, and the task will only run if the function returns `true`:

```
$schedule
    ->command("inspire")
    ->hourly()
    ->when(function () {
        $weather = . . .;
        if ($weather == "sunny") {
            return true;
        } else {
```

```
            return false;
        }
    });
```

The above example schedules a task for execution based on the weather. It only executes on sunny days. The one below uses the `skip()` method to defines condition for skipping a scheduled task.

```
$schedule
    ->command("inspire")
    ->hourly()
    ->skip(function () {
        $weather = . . .;
        if ($weather == "rainy") {
            return true;
        } else {
            return false;
        }
    });
```

Lastly, you may also schedule tasks based on the current application environment, which is helpful if you want to schedule some tasks only in production.

```
$schedule
    ->command("inspire")
    ->weekdays()
    ->environments(["staging", "production"]);
```

In the above example, the task will only run in staging and production environments.

Additional options

The scheduling methods discussed so far should be enough for most scenarios, but Laravel offers some advanced techniques for creating more complex schedules.

1. The `cron()` method

The `cron()` method allows you to create schedules using raw Cron expressions. For example:

```
$schedule->command("inspire")->cron("5 4 * * sun");
```

This code will schedule the `inspire` Artisan command to run at 04:05 every Sunday.

2. The `at()` method

You may have noticed that some of the frequency methods (such as `mondays()`, `quarterly()`, and others) do not allow you to specify an exact time, which can sometimes be limiting. This can be fixed through the `at()` method as shown below:

```
$schedule
    ->command("inspire")
    ->weekly()
    ->wednesdays()
    ->at("13:15"); // Executes every Wednesday at 13:15
```

3. The `timezone()` method

If you'd like to schedule a task in a different timezone, you can also chain a `timezone()` method in your schedule like this:

```
$schedule
    ->command("inspire")
    ->timezone("America/New_York")
    ->at("2:00")
```

Setting up task hooks

Another useful feature that Laravel provides for task scheduling is task hooks. They let you execute some code before or after the task is executed, or if a certain condition is true (such as if the task failed or succeeded).

For example, you can log a message before and after a task is executed as shown below. Please refer to the linked article for more information on logging in Laravel. We assume you've already configured the logging system correctly for demonstration purposes.

```
use Illuminate\Support\Stringable;

$schedule
    ->exec("bash scripts/backup.bash")
    ->before(function () use ($logger) {
        Log::info("The database backup script executed at " .
    })
    ->after(function (Stringable $output) {
        Log::info($output);
    })
    ->everyMinute();
```

If your scheduled job produces some output, you can access it in the `after()` method through the `$output` variable with the `Illuminate\Support\Stringable` type.

A scheduled tasks might succeed or fail, so Laravel also provides the `onSuccess()` and `onFailure()` hooks for dealing with either outcome:

```
$schedule
    ->command(". . .")
    ->before(function () use ($logger) {
        Log::info("The script weather.php executed at " . tim
    })
    ->onSuccess(function (Stringable $output) {
        // The task succeeded...
    })
    ->onFailure(function (Stringable $output) {
```

```
        // The task failed...
    })
    ->everyMinute();
```

Just like the `after()` hook, you can access the output through `Stringable $output`. Note that the `onSuccess()` and `onFailure()` hooks only work for tasks scheduled with the `command()` or `exec()` methods because task failure is detected through a non-zero exit code.

notifying an external service when a scheduled task has began, completed, or failed. Here are the available methods and their signatures:

- `pingBefore($url)`
- `thenPing($url)`
- `pingBeforeIf($condition, $url)`
- `thenPingIf($condition, $url)`
- `pingOnSuccess($url)`
- `pingOnFailure($url)`

A practical example of task scheduling in Laravel

Now that we've covered several Laravel task scheduling essentials, let's look at some practical examples of how it can be useful in a real-life project. We'll automate some tasks for a class management app where teachers can record the student's grades for each class.

Start by cloning the project to your computer using the following command:

```
$ git clone https://github.com/betterstack-community/class-ma
```

Next, change into the `class-management-app` directory:

```
$ cd class-management-app
```

Rename the `.env.example` file at the project root to `.env`:

```
$ mv .env.example .env
```

Afterward, install the required dependencies with `composer`:

```
$ composer install
```

Generate a new `APP_KEY` by running the following Artisan command:

```
$ php artisan key:generate
```

Before starting the development server, ensure all the required modules are enabled in your `php.ini` file:

```
[php.ini]
extension=pdo.so
extension=pdo_sqlite.so
```

```
extension=sqlite.so
. . .
```

The location of your php.ini file can be found by running:

```
$ php -i | grep "Loaded Configuration File"
```

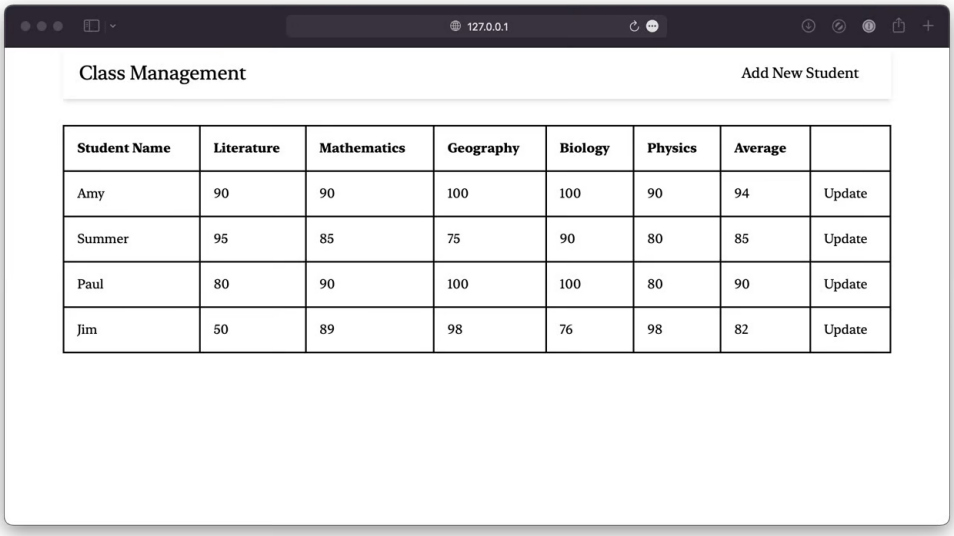
Finally, start the dev server:

```
$ php artisan serve
```

Output

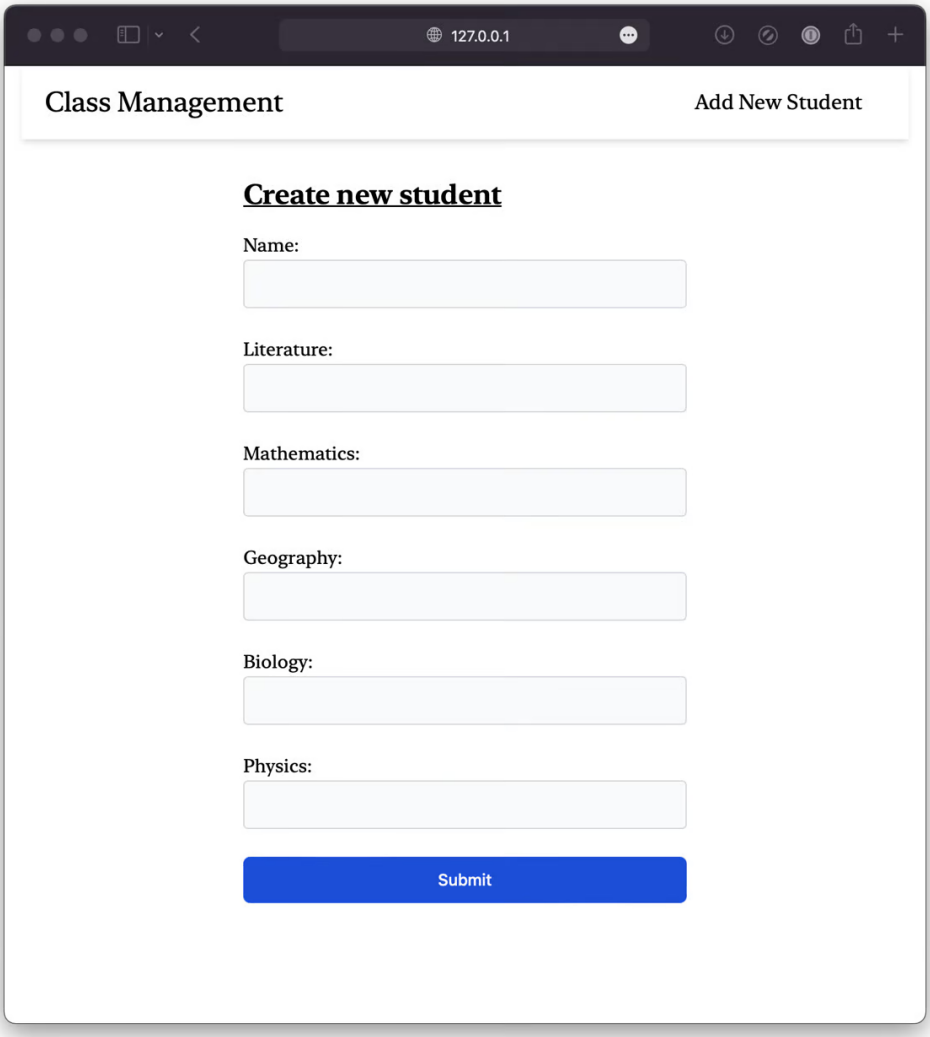
```
INFO  Server running on [http://127.0.0.1:8000].

Press Ctrl+C to stop the server
```

Open your browser and go to http://127.0.0.1:8000. You should see the home page of the class management app. To save time, we've included some dummy data for the project.

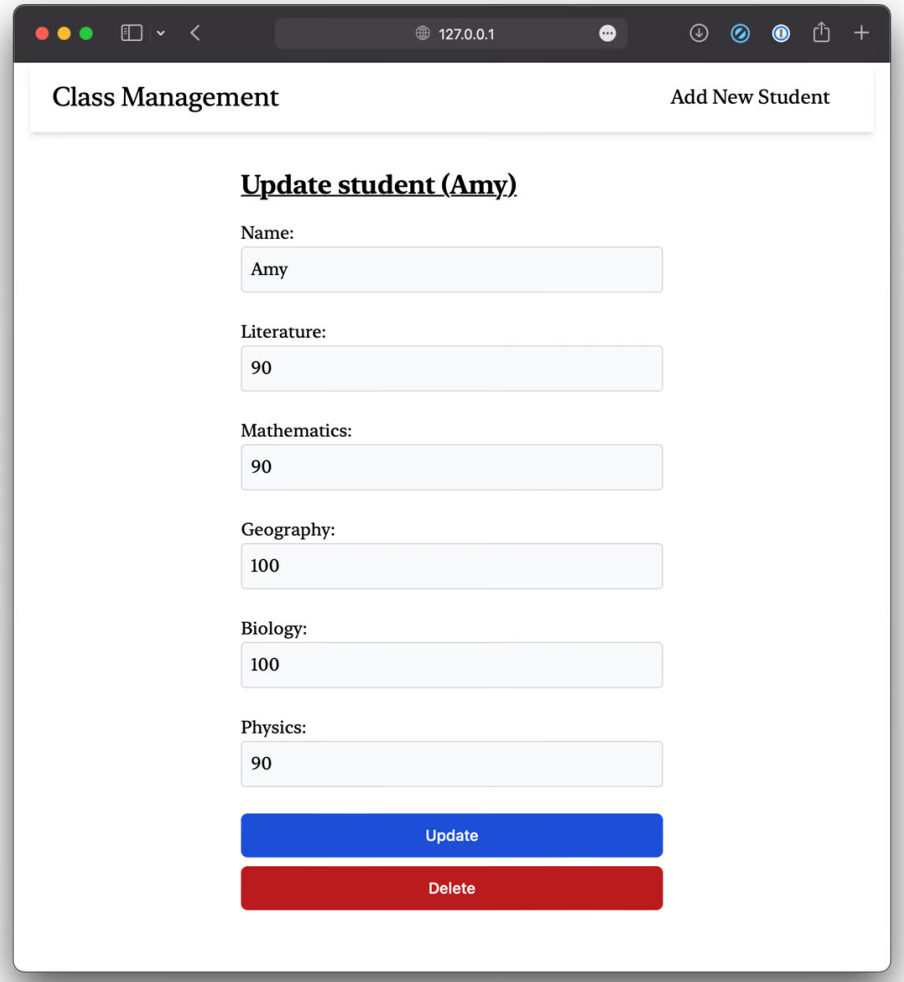


Student Name	Literature	Mathematics	Geography	Biology	Physics	Average	
Amy	90	90	100	100	90	94	Update
Summer	95	85	75	90	80	85	Update
Paul	80	90	100	100	80	90	Update
Jim	50	89	98	76	98	82	Update

You can then create new students:



Or update/delete students:



Automating daily database backups

Let's begin by scheduling a common task that is required in almost all production web applications: a daily database backup. Since the SQLite database is being utilized for this project, backing it up only involves copying the `database.sqlite` file to a specified location.

You can write a shell script to perform this task as shown below:

```
$ mkdir scripts && nano scripts/backup.sh
```

scripts/backup.sh

```
#!/usr/bin/env bash

# Get current date in yyyy-mm-dd format
```

```
NOW=$(date "+%Y-%m-%d")

# If backup directory does not exist, create one
if [ ! -d "backup" ]; then
    mkdir "backup"
fi

# Copy database.sqlite into backup directory
cp "database.sqlite" "backup/${NOW}-database.sqlite"

# Find and delete files older than 7 days
find "backup" -type f -mtime +7 -delete
```

When this script is executed, it will backup the database by copying the `database.sqlite` into the `backup` directory under a new name containing the current date. It also scans for files older than seven days and deletes them to prevent the backups directory from growing too large. This isn't much of a backup, but it suffices to demonstrate the concept of automating such a process.

Go ahead and schedule this script to run once a day with the following code:

app/Console/Kernel.php

```
<?php

namespace App\Console;

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

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $sche
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->exec("bash scripts/backup.sh")->daily();
    }

    /**
     * Register the commands for the application.
     *
     * @return void
```

```
     */
    protected function commands()
    {
        $this->load(__DIR__ . "/Commands");

        require base_path("routes/console.php");
    }
}
```

You can run the `schedule:list` command to see the next time it is scheduled to run:

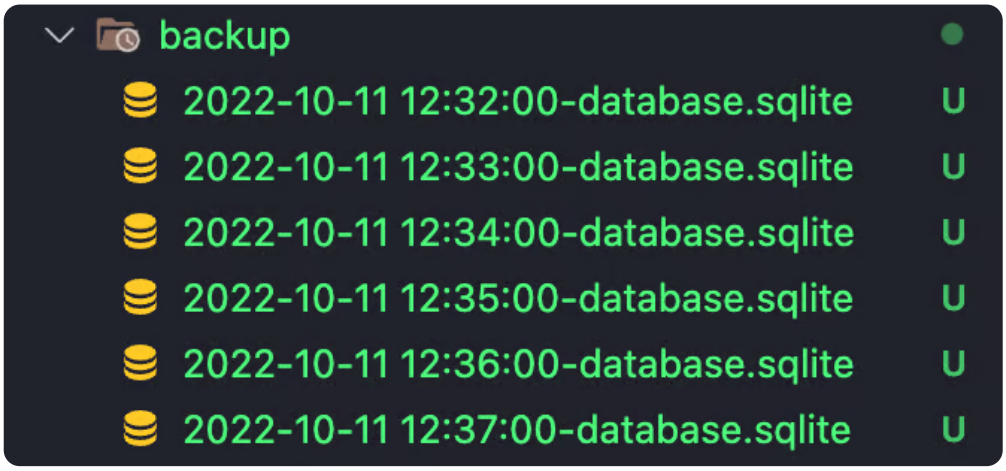
```
$ php artisan schedule:list
```

Output

```
0 0 * * *  bash scripts/backup.sh
.............................. Next Due: 12 hours from now
```

If you want to confirm that the script is working properly, you can change the frequency to from `daily()` to `everyMinute()` and run the `schedule:work` command so that

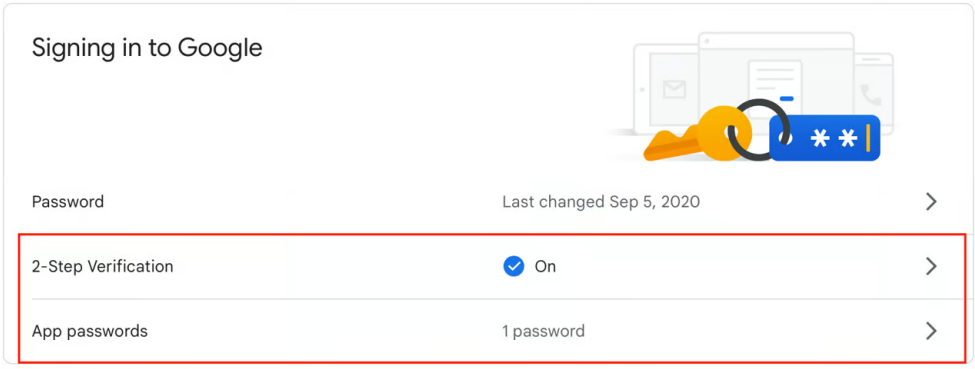
After a minute or so, you should observe the generated `backup` directory as well as the backup files:

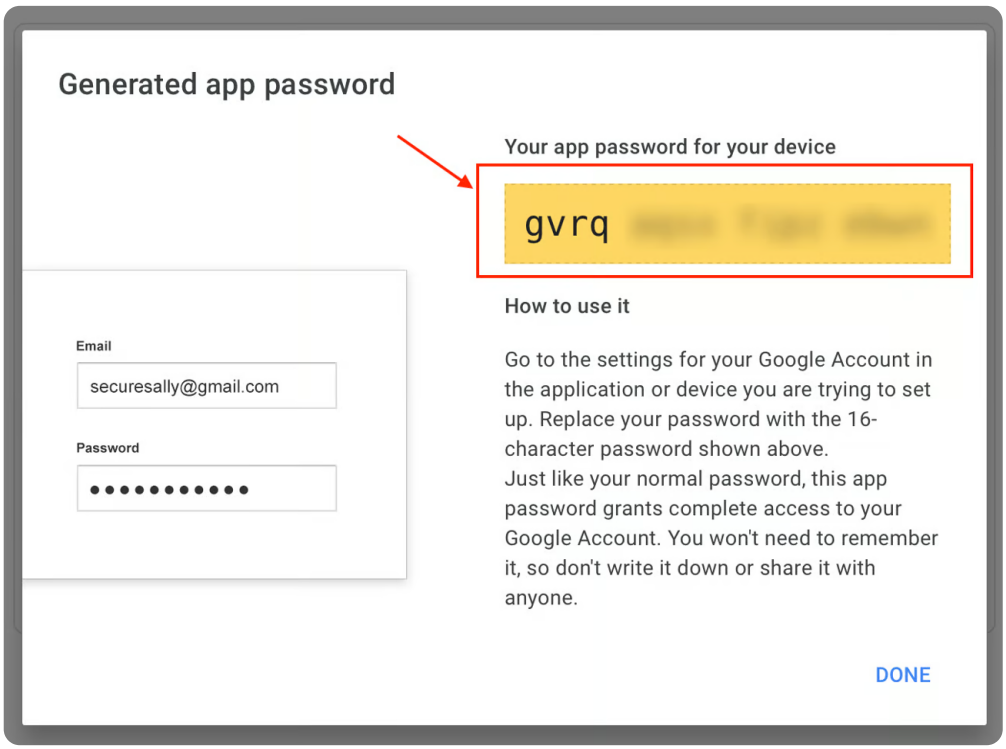


Generating weekly reports

A classic use case for task scheduling is automating the generation of reports for relevant business metrics. In this section, we will demonstrate how to automate such tasks by sending a weekly report of students' performance to an email address. In this example, the report will be in the same format as the application's homepage.

Before proceeding, you must configure Laravel's emailing functionality with Gmail SMTP (or any email provider of your choosing). Start by heading to Google My Account → Security, and enabling 2-Step Verification. Afterward, go to App passwords and create a unique password for your Laravel application. Under **Select app**, choose the **Other (Custom name)** option and type "Class Management App" in the text input.





Once the password is generated, copy it to your clipboard. Head back to your text editor and open your `.env` file once

again. Edit the `MAIL_` section of the file as shown below. Note that the `MAIL_PASSWORD` must be the unique password you just created.

.env

```
. . .
MAIL_DRIVER=smtp
MAIL_HOST=smtp.googlemail.com
MAIL_PORT=465
MAIL_USERNAME=<your_gmail_address>
MAIL_PASSWORD=<your_password>
MAIL_ENCRYPTION=ssl
MAIL_FROM_ADDRESS="<your_gmail_address>"
MAIL_FROM_NAME="${APP_NAME}"
. . .
```

Next, head to your `app/Console/Kernel.php` file, and modify it as shown below. Don't forget to replace the `<your_email>` placeholder with an actual email address.

app/Console/Kernel.php

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\Mail;
use App\Mail\StudentFailClass;
use App\Mail\WeeklyReport;
use App\Models\Student;

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $sche
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        . . .
        $schedule
            ->call(function () {
                Mail::to("<your_email>")->send(
                    new WeeklyReport(Student::all())
```

```
                );
            })
            ->weekly();
    }

    . . .
}
```

The `WeeklyReport` class has not been created yet, so create it by pasting the code below in a new `app/Mail/WeeklyReport.php` file:

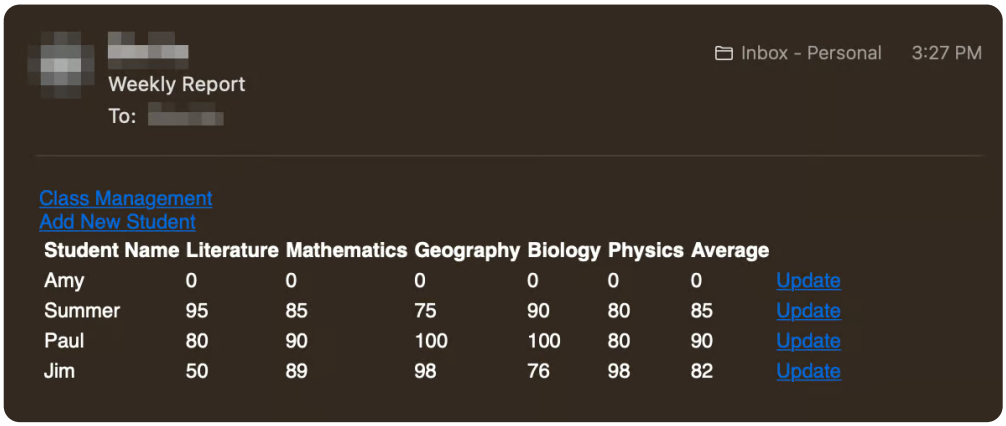
app/Mail/WeeklyReport.php

```php
<?php

namespace App\Mail;

use App\Models\Student;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Mail\Mailables\Envelope;
use Illuminate\Queue\SerializesModels;

class WeeklyReport extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The student instance.
     *
     * @var \App\Models\Student
     */
    public $students;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct($students)
    {
        $this->students = $students;
    }

    /**
     * Get the message envelope.
     *
     * @return \Illuminate\Mail\Mailables\Envelope
```

```
     */
    public function envelope()
    {
        return new Envelope(subject: "Weekly Report");
    }

    /**
     * Get the message content definition.
     *
     * @return \Illuminate\Mail\Mailables\Content
     */
    public function content()
    {
        return new Content(view: "index");
    }

    /**
     * Get the attachments for the message.
     *
     * @return array
     */
    public function attachments()
    {
        return [];
    }
}
```

When the `call()` method is executed, Laravel will send the `index` page (`resources/views/index.blade.php`) to the email of your choice, displaying everyone in the classes and their grade for each subject.

You can temporarily change the schedule frequency to something more suitable for testing such as `everyMinute()` for example, and wait for the task to execute. You should receive the following email in your inbox:



Monitoring scheduled tasks

A simple way to verify that your scheduled tasks are running as expected and providing the correct result is by sending an email to yourself. Assuming your mail credentials have been setup in the `.env` file (as discussed earlier), the `emailOutputTo` method can be used to send the output of task to an email address:

```
$schedule->command("inspire")->everyMinute()->emailOutputTo("
```

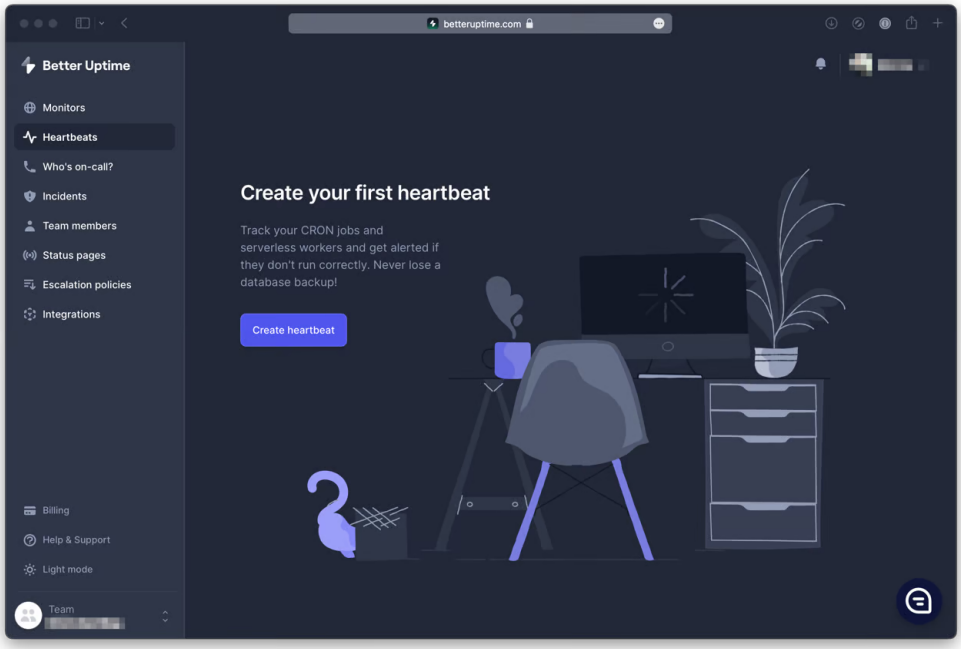
When you run the scheduler, you should receive an email with the output each time the task is executed. If you want to receive an email only when something goes wrong, you can use the `emailOutputOnFailure()` method instead.

```
$schedule->command("inspire")->everyMinute()->emailOutputOnFa
```

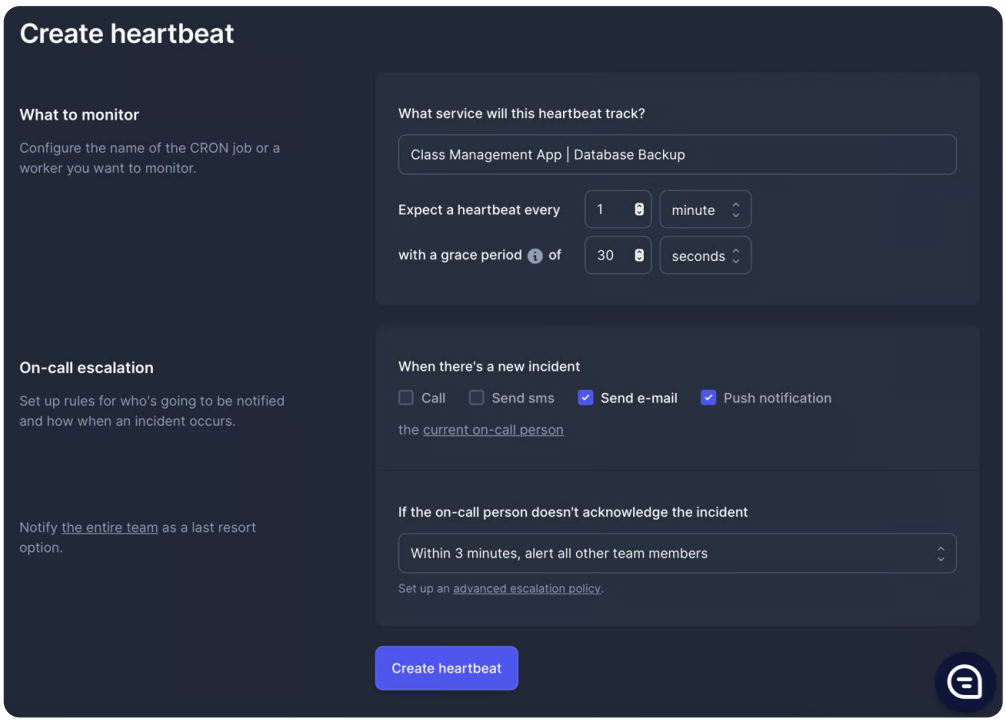
Sending emails via Laravel in the manner described above is a suitable approach for monitoring scheduled tasks in toy projects, especially for non-critical tasks. However, it is unsuitable for production systems where need to guarantee that each task runs according to schedule. A more appropriate solution here is to use a dedicated monitoring service.

Better Uptime is an excellent monitoring and incident management platform that can monitor your entire infrastructure and alert you appropriately if something goes wrong. This section will discuss how to use Better Uptime to monitor the status of your scheduled Laravel tasks.

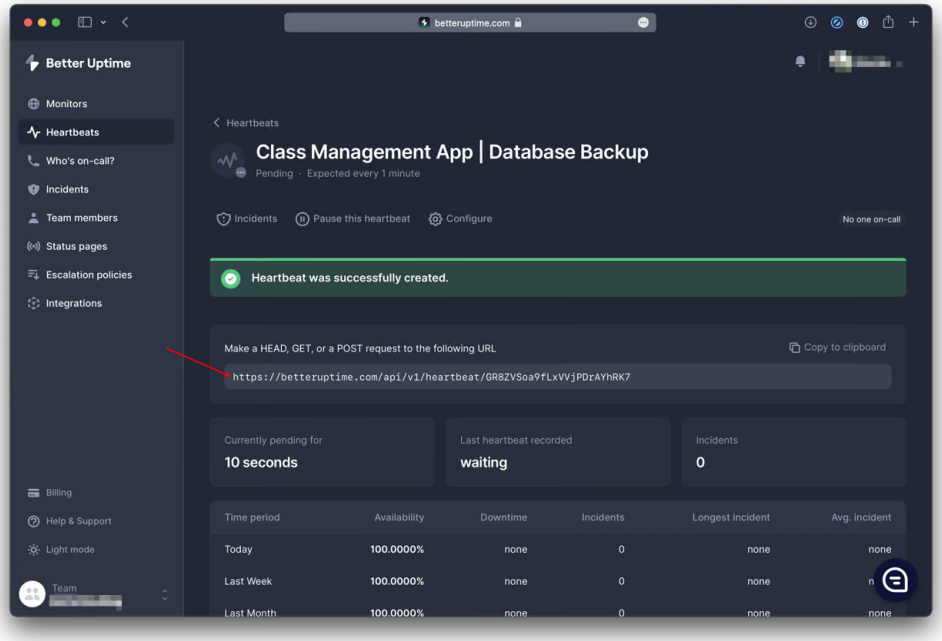
Go ahead and create a free Better Uptime account if you don't have one already. Once you are signed in, head to the **Heartbeats** section and create a new heartbeat.



Choose an appropriate name for your monitor and select how often you expect this job to be repeated. Then, in the **On-call escalation** section, select how you wish to be notified when the job fails to execute.



Once you are done, click **Save Changes**, and you should see this page:



The highlighted URL above is how Better Uptime can monitor your scheduled task. Every time a task executes, you should make a HEAD, GET, or POST request to this URL.

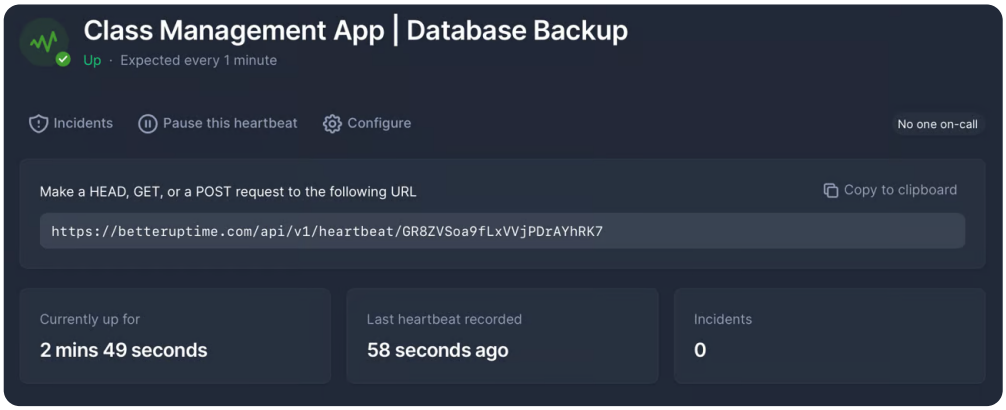
Head back to your `app/Console/Kernel.php` file and add the `thenPing()` hook for the corresponding scheduled task as shown below:

app/Console/Kernel.php

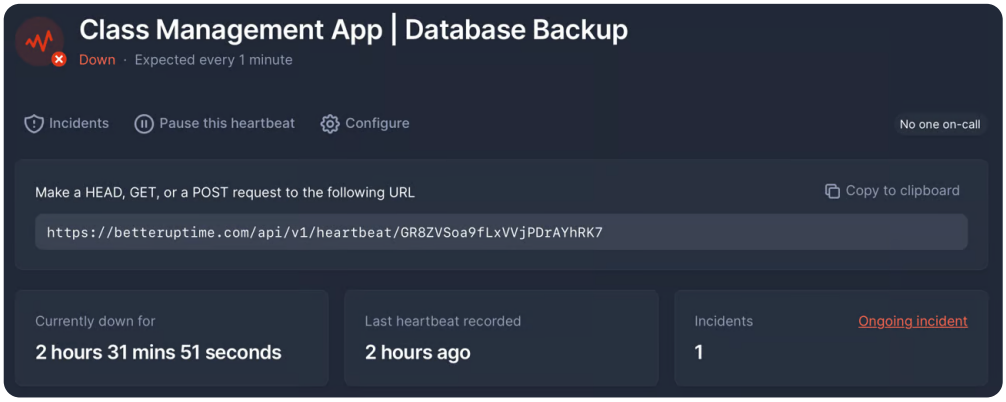
```
<?php

namespace App\Console;

. . .

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $sche
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        . . .

        $schedule
            ->exec("bash scripts/backup.sh")
            ->thenPing("https://betteruptime.com/api/v1/heart
            ->everyMinute();
    }
```

```
    . . .
}
```

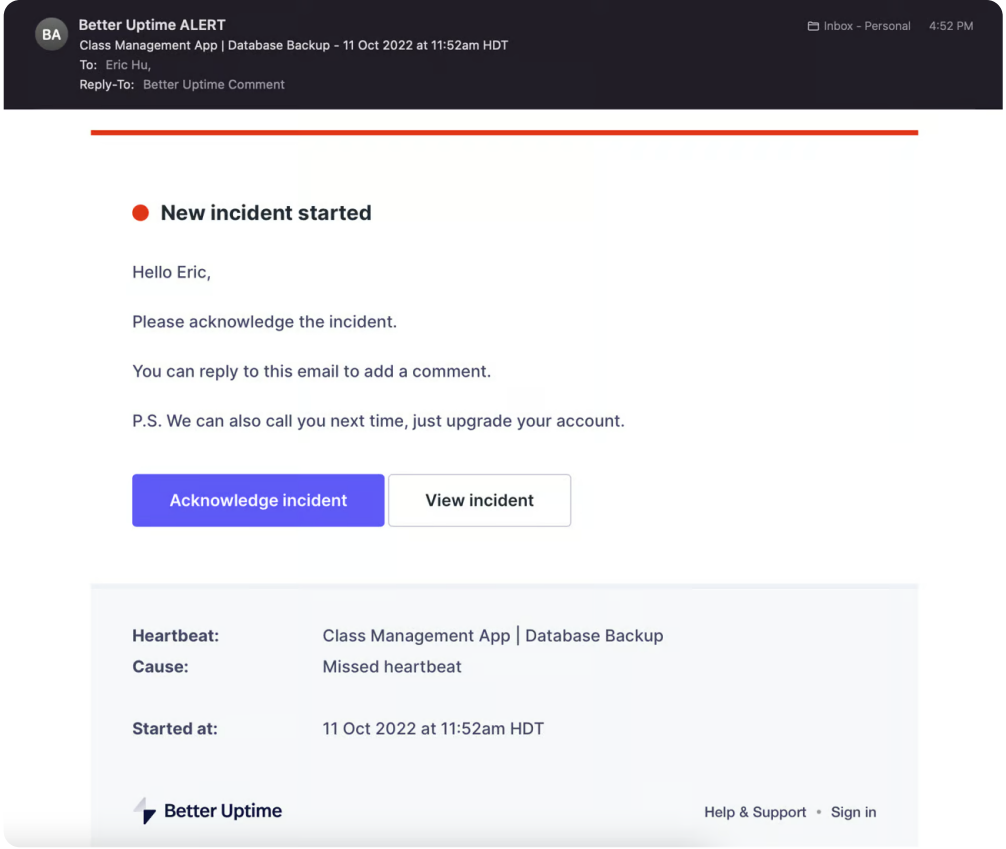
The `thenPing()` method sends a request to the Better Uptime API once the task has finished executing. Once Better Uptime starts receiving requests, the monitor will be marked as "Up", which means the scheduled task is up and running.



You can then simulate an incident by stopping the scheduler with `Ctrl-C`. If Better Uptime does not receive a request within the time frame we've configured, the monitor will be marked as "Down", which means an incident occurred.



You will also receive an alert in the configured channels:



Note that you must create a separate heartbeat for each scheduled task so that they can be monitored independently.

Final thoughts

In this tutorial, we investigated task scheduling in a Laravel application and discussed many of its essential features. We also demonstrated how to set it up in a typical web application, and how to monitor the status of your scheduled tasks so that you are promptly notified if something goes wrong.

If you wish to dig deeper into task scheduling in Laravel, you can read its official documentation for more information.

Thanks for reading, and happy scheduling!

Article by

Eric Hu

Eric is a technical writer with a passion for writing and coding, mainly in Python and PHP. He loves transforming complex technical concepts into accessible content, solidifying

understanding while sharing his own perspective. He wishes his content can be of assistance to as many people as possible.

Got an article suggestion? Let us know

Next article

How to Get Started with Logging in PHP

PHP has built-in features for logging errors but third-party tools also exist for this purpose. How do you know which one to pick? This article will equip you to answer that question.

Make your mark

Join the writer's program

Are you a developer and love writing and sharing your knowledge with the world? Join our guest writing program and get paid for writing amazing technical guides. We'll get

them to the right readers that will appreciate them.

Write for us ›

Writer of the month

Woo Jia Hao

Woo Jia Hao is a software developer from Singapore. He is an avid learner who...

Build on top of Better Stack

Write a script, app or project on top of Better Stack and share it with the world. Make a public repository and share it with us at our email.

community@betterstack.com

or submit a pull request and help us build better products for everyone.

See the full list of amazing projects on github

Solutions

Log management

Uptime monitoring

Incident management

Status page

Resources

Help & Support

Uptime docs

Logs docs

Company

Compare

Pingdom

Pagerduty