# How to contribute to Lighthouse

Thank you for contributing to Lighthouse. Here are some tips to make this easy for you.

## The process

1. Fork the project

1. Create a new branch

1. Code, commit and push

1. Open a pull request detailing your changes. Make sure to follow the [template](.github/PULL\_REQUEST\_TEMPLATE.md)

## Setup

The project setup is based upon [docker-compose](https://docs.docker.com/compose/install/).

For convenience, common tasks are wrapped up in the [Makefile](Makefile) for usage with [GNU make](https://www.gnu.org/software/make/).

Just clone the project and run the following in the project root:

make setup

make

To see the available commands, run:

make help

## Testing

We use \*\*PHPUnit\*\* for unit tests and integration tests.

Have a new feature? You can start off by writing some tests that detail

the behaviour you want to achieve and go from there.

Fixing a bug? The best way to ensure it is fixed for good and never comes

back is to write a failing test for it and then make it pass. If you cannot

figure out how to fix it yourself, feel free to submit a PR with a failing test.

Here is how to set up Xdebug in PhpStorm https://www.jetbrains.com/help/phpstorm/configuring-xdebug.html

> Enabling Xdebug slows down tests by an order of magnitude.

> Stop listening for Debug Connection to speed it back up.

Set the environment variable `XDEBUG\_REMOTE\_HOST` to the IP of your host machine as

seen from the Docker container. This may differ based on your setup: When running

Docker for Desktop, it is usually `10.0.2.2`, when running from a VM it is something else.

## Documentation

The documentation for Lighthouse is located in [`/docs`](/docs).

You can check out the [Docs README](/docs/.github/README.md) for more information on how to contribute to the docs.

## Changelog

We keep a [changelog](/CHANGELOG.md) to inform users about changes in our releases.

When you change something notable, add it to the top of the file in the `Unreleased` section.

Choose the appropriate type for your change:

- `Added` for new features.

- `Changed` for changes in existing functionality.

- `Deprecated` for soon-to-be removed features.

- `Removed` for now removed features.

- `Fixed` for any bug fixes.

- `Security` in case of vulnerabilities.

Then, add a short description of your change and close it off with a link to your PR.

## Code guidelines

### `protected` over `private`

Always use class member visibility `protected` over `private`. We cannot foresee every

possible use case in advance, extending the code should remain possible.

### Laravel feature usage

We strive to be compatible with both Lumen and Laravel.

Do not use Facades and utilize dependency injection instead.

Not every application has them enabled - Lumen does not use Facades by default.

Prefer direct usage of Illuminate classes instead of helpers.

```php

// Correct usage

use \Illuminate\Support\Arr;

Arr::get($foo, 'bar');

// Wrong usage

array\_get($foo, 'bar');

```

A notable exception is the `response()` helper - using DI for injecting a

`ResponseFactory` does not work in Lumen, while `response()` works for both.

### Type definitions

Prefer the strictest possible type annotations wherever possible.

If known, add additional type information in the PHPDoc.

```php

/\*\*

\* We know we get an array of strings here.

\*

\* @param array<string> $bar

\* @return string

\*/

function foo(array $bar): string

```

For aggregate types such as the commonly used `Collection` class, use

the generic type hint style. While not officially part of PHPDoc, it is understood

by PhpStorm and most other editors.

```php

/\*\*

\* Hint at the contents of the Collection.

\*

\* @return \Illuminate\Support\Collection<string>

\*/

function foo(): Collection

```

Use `self` to annotate that a class returns an instance of itself (or its child).

Use [PHPDoc type hints](http://docs.phpdoc.org/guides/types.html#keywords) to

differentiate between cases where you return the original object instance and

other cases where you instantiate a new class.

```php

<?php

class Foo

{

/\*\*

\* Some attribute.

\*

\* @var string

\*/

protected $bar;

/\*\*

\* Use $this for fluent setters when we expect the exact same object back.

\*

\* @param string $bar

\* @return $this

\*/

public function setBar(string $bar): self

{

$this->bar = $bar;

return $this;

}

/\*\*

\* Use static when you return a new instance.

\*

\* @return static

\*/

public function duplicate(): self

{

$instance = new static;

$instance->bar = $this->bar;

return $instance;

}

}

```

### Annotating Exception Throwing

Only annotate `@throws` for Exceptions that are thrown in the function itself.

```php

/\*\*

\* @throws \Exception

\*/

function foo(){

throw Excection();

}

/\*\*

\* No need to annotate the Exception here, even though

\* it is thrown indirectly.

\*/

function bar(){

foo();

}

```

## Code style

We use [StyleCI](https://styleci.io/) to ensure clean formatting, oriented

at the Laravel coding style.

Prefer explicit naming and short, focused functions over excessive comments.

### Alignment

Do not align stuff horizontally, it leads to ugly diffs.

```php

// Right

[

'foo' => 1,

'barbaz' => 2,

]

// Wrong

[

'foo' => 1,

'barbaz' => 2,

]

```

### Multiline Ternary Expressions

Ternary expressions must be spread across multiple lines.

```php

$foo = $cond

? 1

: 2;

```

### Class References

When used in the actual source code, classes must always be imported at the top.

Class references in PHPDoc must use the full namespace.

```php

<?php

use Illuminate\Database\Eloquent\Model;

class Foo

{

/\*\*

\* @var \Illuminate\Database\Eloquent\Model

\*/

protected $model;

public function bar(): Model

{

return $this->model;

}

}

```

You can use the following two case-sensitive regexes to search for violations:

```regexp

@(var|param|return|throws).\*\|[A-Z]

@(var|param|return|throws)\s\*[A-Z]

```

### Test Data Setup

Use relations over direct access to foreign keys.

```php

$user = factory(User::class)->create();

// Right

$post = factory(Post::class)->make();

$user->post()->save();

// Wrong

$user = factory(Post::class)->create([

'user\_id' => $post->id,

]);

```

Use properties over arrays to fill fields.

```php

// Right

$user = new User();

$user->name = 'Sepp';

$user->save();

// Wrong

$user = User::create([

'name' => 'Sepp',

]);

```

## Benchmarks

We use [phpbench](https://github.com/phpbench/phpbench) for running benchmarks

on performance critical pieces of code.

Run the reports that are defined in `phpbench.json` via the command line,

for example:

vendor/bin/phpbench run --report=ast