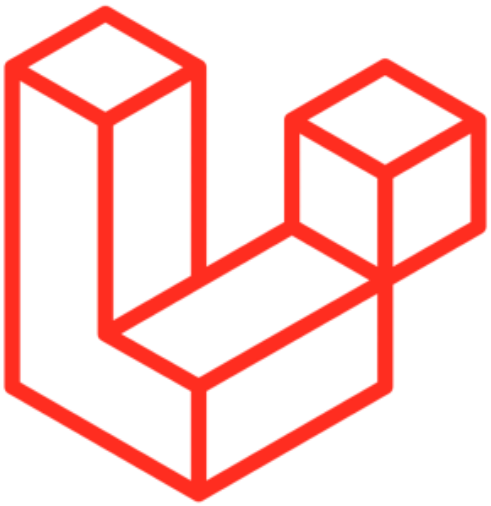
LARAVEL MODULAR



Poscar

Table of Contents

Laravel is a server-side PHP framework, with it you can build full-stack applications, meaning applications with features typically requiring a back-end, such as accounts, exports order management, etc. It's an entirely server-side framework that manages data with the help of Model-View-Controller (MVC) design which breaks an application back-end architecture into logical parts. Laravel have packages for us to use in applications, if you work on big projects, you can use Laravel Modules to organize our code into smaller modules.

1. Laravel Modular

Laravel Modular is a Laravel package that was created to manage large Laravel apps using modules. A module is like a Laravel package, it has some views, models, controllers, and migrations. It comes with a folder structure and feature is organized in a nice directory structure as below:

- We use Laravel Modules because it easy to organize our code into smaller modules. We know exactly where our features/modules are and we also know where their controllers, models, views, and routes are placed. When we work on bigger or medium-level Laravel projects we always work on different features.

-To organize our code in a feature or module-based approach where we can bundle our feature or module.

Example:

- Controller
- Models
- Migrations
- Views
- And other folders we might need.

2. Install Laravel Modules package

-Install Laravel project => go to cmd and write the command below:

composer create-project laravel/laravel Blog --prefer-dist

- Before we use Laravel Modules, we have to Install and setup composer first.

Composer

To install through Composer, by run the following command:

composer require nwidart/laravel-modules

The package will automatically register a service provider.

Optionally, publish the package's configuration file by running:

php artisan vendor:publish --provider="Nwidart\Modules\LaravelModulesServiceProvider"

Autoloading

By default, the module classes are not loaded automatically. You can autoload your modules using psr-4

"Modules\\": "Modules/"

```
"autoload": {
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/",
        "Modules\\": "Modules/"
    }
},
```

Main app composer.json file

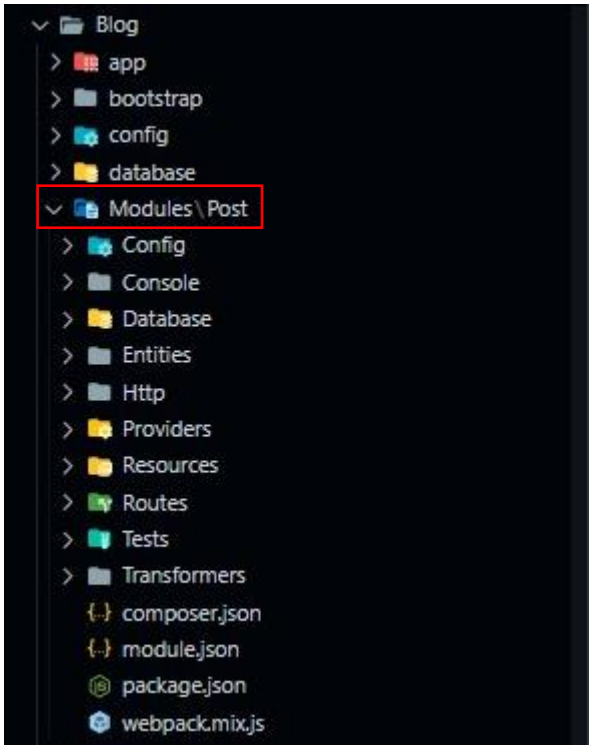
Don’t forget run this command:

composer dump-autoload

3. Create internal custom module

- Generate a new module:

php artisan module:make Post

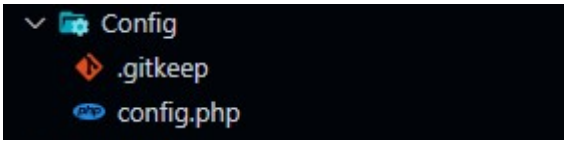


Project folder

Directory Structure

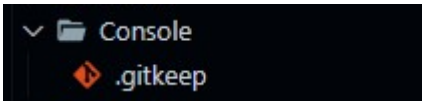
➢ The Config directory

The config directory, as the name implies, contains all of your application's configuration files. It's a great idea to read through all of these files and familiarize yourself with all of the options available to you.



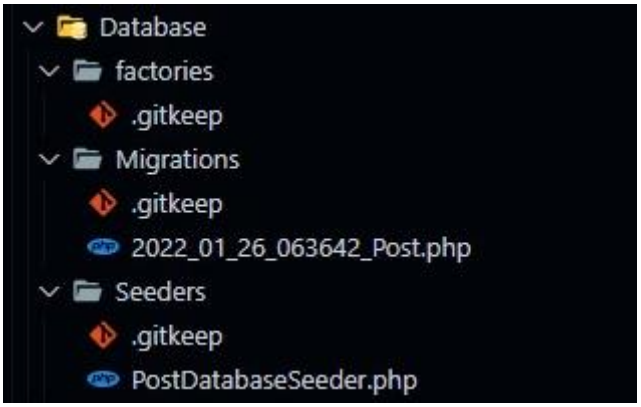
➢ The Console directory

The Console directory contains all of the custom Artisan commands for your application. These commands may be generated using the make:command command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your scheduled tasks are defined.



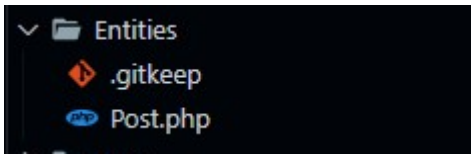
➢ The Database directory

The database directory contains your database migrations, model factories, and seeds. If you wish, you may also use this directory to hold an SQLite database.



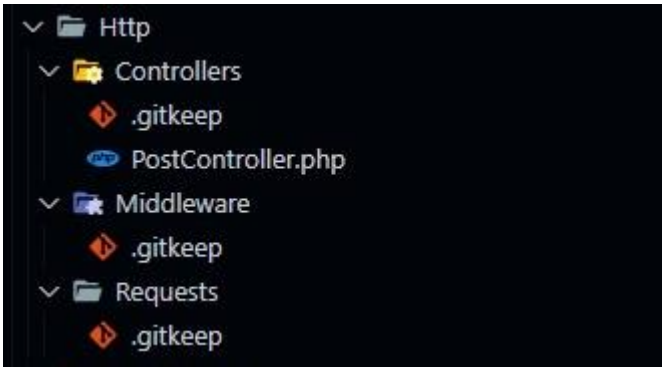
➢ The Entities directory

The entities directory contains all of your Eloquent model classes. The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Entities" which is used to interact with that table. Entities allow you to query for data in your tables, as well as insert new records into the table.



➢ The Http directory

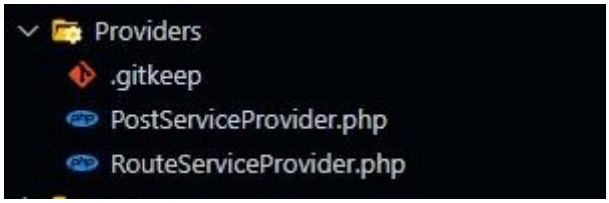
The Http directory contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.



➢ The Providers directory

The Providers directory contains all of the service providers for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.

In a fresh Laravel application, this directory will already contain several providers. You are free to add your own providers to this directory as needed.



➢ The Resources directory

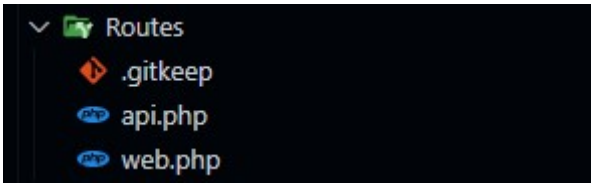
The resources directory contains your views as well as your raw, un-compiled assets such as CSS or JavaScript. This directory also houses all of your language files.

➢ The Routes directory

The routes directory contains all of the route definitions for your application. By default, several route files are included with Laravel: web.php, api.php, console.php, and channels.php.

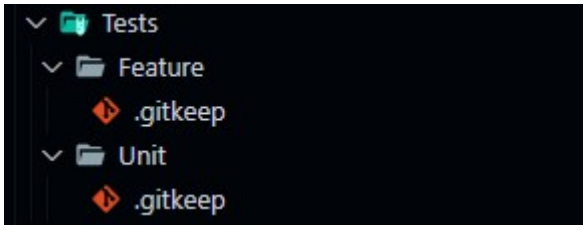
The web.php file contains routes that the RouteServiceProvider places in the web middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then it is likely that all of your routes will most likely be defined in the web.php file.

The api.php file contains routes that the RouteServiceProvider places in the api middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.



➢ The Tests directory

The tests directory contains your automated tests. Example PHPUnit unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word Test. You may run your tests using the phpunit or php vendor/bin/phpunit commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the php artisan test Artisan command.



➢ Composer.json File

Any of these packages may be used with Laravel by requiring them in you composer.json file. Instead of requiring users to manually add your service provider to the list, you may define the provider in the extra section of your package's composer.json file. In addition to service providers, you may also list any facades you would like to be registered:

```
{
    "name": "nwidart/post",
    "description": "",
    "authors": [
        {
            "name": "Nicolas Widart",
            "email": "n.widart@gmail.com"
        }
    ],
    "extra": {
        "laravel": {
            "providers":
            [
                "Barryvdh\\Debugbar\\ServiceProvider"
            ],
            "aliases": {
                "Debugbar": "Barryvdh\\Debugbar\\Facade"
            }
        }
    },
    "autoload": {
        "psr-4": {
            "Modules\\Post\\": ""
        }
    }
}
```

Once your package has been configured for discovery, Laravel will automatically register its service providers and facades when it is installed, creating a convenient installation experience for your package's users.

➢ Module.json File

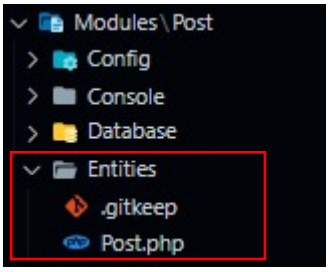
Note that every module has a module.json file, in which you can enable and disable the module.

```
{
    "name": "Post",
    "alias": "post",
    "description": "",
    "keywords": [],
    "priority": 0,
    "providers": [
        "Modules\\Post\\Providers\\PostServiceProvider"
    ],
    "aliases": {},
    "files": [],
    "requires": []
}
```

Artisan commands

- Generate multiple modules at once:

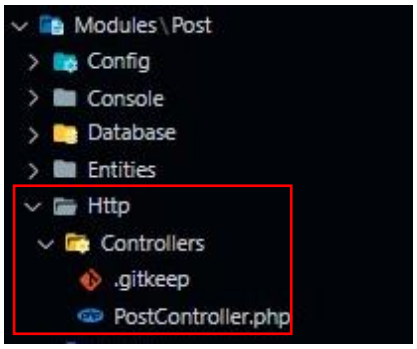
php artisan module:make-model Post Post



- Controller:

php artisan module:make-controller PostController Post

>> For this we don't need to create. It will create automatically.



- Migrate table:

php artisan module:make-migration Post Post

Migrate the given module, or without a module an argument, migrate all modules.

php artisan module:migrate Post

- Publish Modules:

-***php artisan module:publish***

OR

- php artisan vendor:publish --tag=courier-config

4. Create external custom module

This step is nothing different from the previous step by just following the previous step to create a new Laravel project, and modules. However, in this step, we have to create the package for publishing the composer repository. Because we want to use this package with another project by just importing it into the composer.

Create Laravel package

Many developers make use of **REUSABLE** code by copying and pasting classes from one project to another. To make us easier we can build our own packages. In Laravel framework, we can write libraries where we can just call its services without worrying about how it was implemented. We can reuse these libraries and manage the separately from our projects main source code. These are call Laravel package.

What are packages?

Packages are stand-alone source code which developer can import into their projects using a package management tool. Instead of copying and pasting the libraries source code to another project, developers only need to import the package and call any service that it provides to help with their project development.

Why should we use packages?

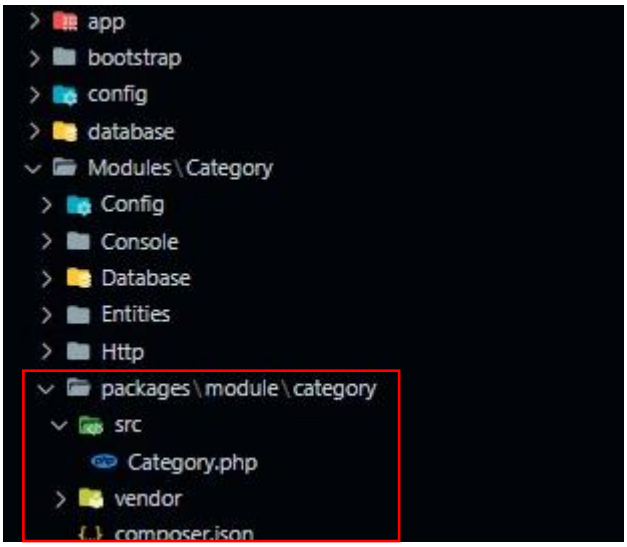
When we have lots of projects, where some functionalities or features are similar, it would be better if we start implementing these features in packages. When you have a team or an individual maintaining a package, let's say GITLAB, it is much easier to collaborate possible fixes to bugs or improvements. This means that each package will undergo the same process as a normal project.

How do we manage Laravel packages?

Composer is dependency management tool for PHP packages. It helps us to manage our packages by pulling in all dependencies that each package requires without duplicating them in your vendor directory. You can define the version of the package you want and even set to automatically update to the latest version every time you run your package updates.

File Structure

On the default folder structure of a new Laravel project, create a file structure as shown below t house our package development.



Note that inside the packages folder should be the vendor's name module followed by the package name category. The src folder will contain all of our package source code. In some instances, you may also create a dist branch if you are compiling your assets for a production server. In this way, developers that will be using your package will have an option to choose between the source files in the src folder or the distribution files in the dist folder when installing your package through Composer.

Package composer.json

In your command prompt, navigate to packages/module/category directory. Then we will need to initialize this folder as a composer package by running the following command:

```
$ composer init
```

This command will ask you for some info regarding your package. You can accept the defaults by pressing enter and edit composer.json file that will be created later.

```
PS C:\xampp\Xampp\htdocs\event-me\Event-Me\Modules\Category\packages\module\category>
composer init


  Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [lenovo/category]: module/category
Description []: Lyden a simple Laravel package.
Author [lydenchai <lyeden999@gmail.com>, n to skip]:
Minimum Stability []: dev
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []: MIT

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Add PSR-4 autoload mapping? Maps namespace "Module\Category" to the entered relative path.
[src/, n to skip]:

{
    "name": "module/category",
    "description": "Lyden a simple Laravel package.",
    "type": "project",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "Module\\Category\\": "src/"
        }
    },
    "authors": [
        {
            "name": "lydenchai",
            "email": "lyeden999@gmail.com"
        }
    ],
    "minimum-stability": "dev",
    "require": {}
}

Do you confirm generation [yes]? y
Generating autoload files
Generated autoload files
PSR-4 autoloading configured. Use "namespace Module\Category;" in src/
Include the Composer autoloader with: require 'vendor/autoload.php';
```

After the initialization, we can edit the composer.json file as follows.

```
{
    "name": "module/category",
    "description": "Lyden a simple Laravel package.",
    "type": "project",
    "license": "MIT",
    "authors": [
        {
            "name": "lydenchai",
            "email": "lyeden999@gmail.com"
        }
    ],
    "minimum-stability": "dev",
    "autoload": {
        "psr-4": {
            "Module\\Category\\": "src/"
        }
    },
    "require": {}
}
```

Package/module/category/composer.json

The only thing added here and that should be noted is the autoload object property. This will automatically load your package with the given namespace Module\Post. You may also include any package dependencies in the require property.

Project composer.json

Autoload the namespace for your package by adding the line "Module\\Post\\": “packages/post /ost/src" inside the autoload->psr-4 property of your project composer.json file. With this, your package classes inside the src folder will be autoloaded and available for our project to use.

```
"autoload": {
        "psr-4": {
            "Module\\Category\\": "packages/module/category/src/"
        },
        "classmap": [
            "database/seeds",
            "database/factories"
        ]
    },
```

Main app composer.json file

There are other options for package discovery which you can reference here. Now, since we’ve edited our project’s composer.json file, we should reload it by running the following command in the project root directory:

```
$ composer dump-autoload
```

Category.php Class

Create the category.php class inside the src folder as follows.

```
<?php

namespace Module\Category;

class Category
{
    public function category(String $sName)
    {
        return 'Hi ' . $sName . '! How are you doing today?';
    }
}
```

packages/module/category/src/Category.php

We should indicate a namespace for this class which should be the same namespace that you indicated in the project composer.json file for it to be autoloaded. In our case, following namespace convention in Laravel, it should be Module\Category.

Testing Our Package

Our simple package is now ready for testing. But first, let's create a route and call our package Category class from this route for now.

```
<?php
use Module\Category\Category;
use Illuminate\Support\Facades\Route;

Route::get('/category/{name}', function($sName) {
    $oCategory = new Category();
    return $oCategory->category($sName);
});
```

Routes/web.php

We can import our Category class from our package by specifying its namespace first with the use keyword. Then we can instantiate the class and call its member functions in our project's route file.

Run an out of the box development server from Laravel by running the following command in your project root directory:

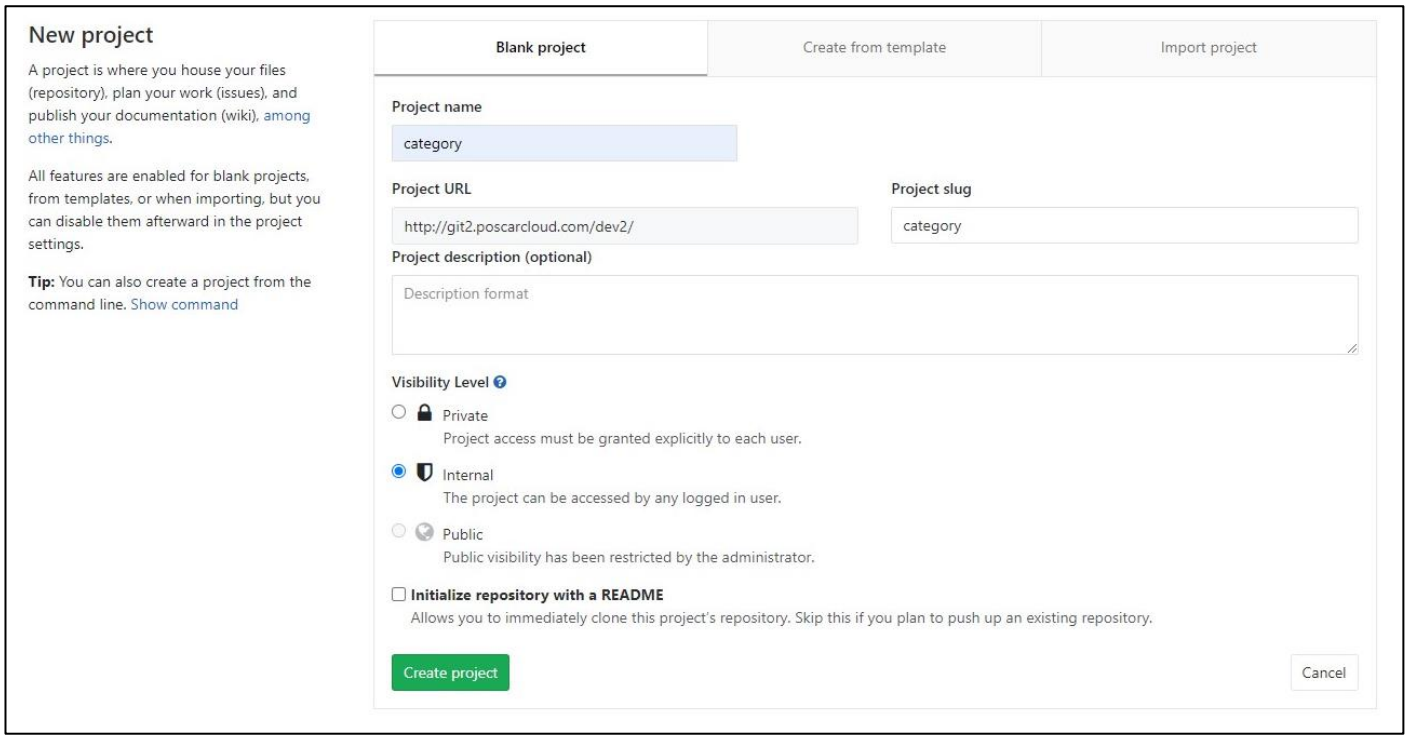
```
$ php artisan serve
Laravel development serve stated: http://127.0.0.1:8000
```

5. Publish custom module composer repository

Publishing Packages to GitLab

Creating GitLab repository

First, we should create a new project in GitLab. Make sure the visibility level is set to public. We may also set it to private if we want control over who can use our package. Doing this will require additional steps. But for this tech blog's purposes, we will set it as public.



Navigate to our package directory packages/module/post. Initialize git, commit changes, and tag changeset with semantic versioning by running the following commands.

```
// packages/module/category

$ git init
$ git checkout -b master
$ git add .
$ git commit -m "initial commit"
$ git tag 1.0.0
```

Push local source code to remote GitLab serve

Make sure your account is already setup on your local machine for us to push our source code to the GitLab server. We only need to push the packages/module/post directory to GitLab. It is advisable to include files like CHANGELOG.md and readme.md for other developers to know more about your package.

To push our local source code to the newly created GitLab project, run the following commands. Make sure to replace the origin URL with your own remote repository URL.

```
// packages/module/category

$ git remote add origin https://git2.poscarcloud.com/dev2/category.git
$ git push -u origin --all
$ git push -u origin --tags
```

Now, we are all set. Other developers may install our package to their own projects to speed up their development time. In the next section, we will try to import our package in a new Laravel project.

Import Laravel Packages

Now we can import our package from the GitLab server into a new Laravel project using Composer. Create another Laravel project on your local machine.

Require our package in composer.json

Edit the new project's composer.json file to require our package and also indicate our repository in GitLab for Composer to know where to fetch packages aside from the default packages repositories.

```
"require": {
    "php": "^7.1.3",
    "fideloper/proxy": "^4.0",
    "laravel/framework": "5.7.*",
    "laravel/tinker": "^1.0",
    "module /category ": "^1.0.0"
},
"repositories": [
    {
        "type": "vcs",
        "url": " https://git2.poscarcloud.com/dev2/category"
    }
]
```

Main app composer.json file

The repository's property lists all non-packagist repositories. If you are going to install several packages from the same GitLab domain, instead of specifying each package's repository you may use the type "composer" and only indicate the domain in the URL. However, there's an additional setup that your GitLab admin should do in this case. For now, let's stick to a single repository with type "vcs".

Run the following command to load the changes we made to our project's composer.json file and install our package.

```
$ composer update
```

We now should find our package in the vendor directory of our project. We should be able to use our package like any other packages that we require in our project. We can test our package by just following the previous section on testing our package but this time do it on the new Laravel project that you just created.

6. Attachments

1. https://nwidart.com/laravel-modules/v6/introduction
2. https://laravel.com/docs/8.x/structure
3. https://medium.com/@francismacugay/build-your-own-laravel-package-in-10-minutes-using-composer-867e8ef875dd

Thank you!