DAVID CARR

LARAVEL MODULES

V11

Table of Contents

Introduction

`nwidart/laravel-modules` is a Laravel package which was created to manage your large Laravel app using modules. A module is like a Laravel package, it has some views, controllers or models. This package is supported and tested in Laravel 11.

Quick Example

Generate your first module using `php artisan module:make Blog`. The following structure will be generated.

```
├── app
│   ├── Http
│   │   └── Controllers
│   │       └── BlogController.php
│   ├── Models
│   └── Providers
│       ├── BlogServiceProvider.php
│       └── RouteServiceProvider.php
├── config
│   └── config.php
├── database
│   ├── factories
│   ├── migrations
│   └── seeders
│       └── BlogDatabaseSeeder.php
├── resources
│   ├── assets
│   │   ├── js
│   │   │   └── app.js
│   │   └── sass
│   │       └── app.scss
│   └── views
│       ├── layouts
│       │   └── master.blade.php
│       └── index.blade.php
├── routes
│   ├── api.php
│   └── web.php
├── tests
│   ├── Feature
│   └── Unit
├── composer.json
├── module.json
├── package.json
└── vite.config.js
```

Requirements

The modules package requires **PHP 8.2** or higher. The Laravel package also requires **Laravel 11.0**.

Changelog

All Notable changes to `laravel-modules` will be documented in the changelog file on the module repository.

https://github.com/nWidart/laravel-modules/blob/master/CHANGELOG.md

Installation and Setup

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 and publish stubs by running:

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

To publish only the config:

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

To publish only the stubs

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

> **Notice:** From V10.0.3

To publish only vite-modules-loader.js

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

Autoloading

from v11.0 autoloading `"Modules\\": "modules/",` is no longer required.

By default, the module classes are not loaded automatically. You can autoload your modules by adding merge-plugin to the extra section:

```
"extra": {
    "laravel": {
        "dont-discover": []
    },
    "merge-plugin": {
        "include": [
            "Modules/*/composer.json"
        ]
    }
},
```

Tip: don't forget to run `composer dump-autoload` afterwards

Upgrade

Heads up: If you upgrade to v6 from the previous version, run the following command: `php artisan module:v6:migrate`

Upgrading from v8.3.0

If you have an existing config file, and you get an error:

```
Target class [CommandMakeCommand] does not exist
```

replace the commands array with:

```
 'commands' =>
\Nwidart\Modules\Providers\ConsoleServiceProvider::defaultCommands()
        ->merge([
            // New commands go here
        ])->toArray(),
```

Composer Merge Plugin

The first time you upgrade to v11 you will be asked whether to enable the merge plugin, press y to allow. It's now required for merging composer files from modules.

```
Do you trust "wikimedia/composer-merge-plugin" to execute code and wish
to enable it now? (writes "allow-plugins" to composer.json) [y,n,d,?]
```

Tip: run `composer dump-autoload` after any composer changes

Composer update

There is a new command `php artisan module:composer-update` to update all module's composer.json files.

This will update autoloading paths for existing modules.

Config

Please update your `config/modules.php` file the generator paths have been updated as well as using internal paths for commands.

Please note the new paths only affect new modules / generated files.

The easiest way is to delete the file and re-publish it:

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

Autoloading

from v11.0 autoloading `"Modules\\": "modules/",` is no longer required.

Please delete the Modules autoloading section:

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

By default, the module classes are not loaded automatically. You can autoload your modules by adding merge-plugin to the extra section:

```
"extra": {
    "laravel": {
        "dont-discover": []
    },
    "merge-plugin": {
        "include": [
            "Modules/*/composer.json"
        ]
    }
},
```

Modules composer.json files for newly generated modules will contain:

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

This allows all classes to be autoloaded from a new folder called app without requiring App to be in the classes' namespaces.

Existing modules that don't contain an App folder can continue to use their autoloading path:

Autoload all classes to the root of the module.

```
"autoload": {
    "psr-4": {
        "Modules\\Blog\\": ""
    }
}
```

Existing modules that do contain an App folder will need to adjust the autoloading path:

Autoload all classes to the root of the module.

Please change `"Modules\\Blog\\": "app/",` to point to the root of the module:

```
"autoload": {
    "psr-4": {
        "Modules\\Blog\\": "",
        "Modules\\Blog\\Database\\Factories\\": "database/factories/",
        "Modules\\Blog\\Database\\Seeders\\": "database/seeders/"
    }
},
"autoload-dev": {
    "psr-4": {
        "Modules\\Blog\\Tests\\": "tests/"
    }
}
```

Module Structure

Newly generated modules will now have this structure

```
Modules
    └── Blog
        ├── app
        │   ├── Http
        │   │   └── Controllers
        │   │       └── BlogController.php
        │   ├── Models
        │   └── Providers
        │       ├── BlogServiceProvider.php
        │       └── RouteServiceProvider.php
        ├── config
        │   └── config.php
        ├── database
        │   ├── factories
        │   ├── migrations
        │   └── seeders
        │       └── BlogDatabaseSeeder.php
        ├── resources
        │   ├── assets
        │   │   ├── js
        │   │   │   └── app.js
        │   │   └── sass
        │   │       └── app.scss
        │   └── views
        │       ├── layouts
        │       │   └── master.blade.php
        │       └── index.blade.php
        ├── routes
        │   ├── api.php
        │   └── web.php
        ├── tests
        │   ├── Feature
        │   └── Unit
        ├── composer.json
        ├── module.json
        ├── package.json
        └── vite.config.js
```

This can be changed by editing the generator paths in `config/modules.php`

Lumen

Lumen doesn't come with a vendor publisher. In order to use laravel-modules with lumen you have to set it up manually.

Create a config folder inside the root directory and copy `vendor/nwidart/laravel-modules/config/config.php` to that folder named `modules.php`

```
mkdir config
cp vendor/nwidart/laravel-modules/config/config.php config/modules.php
```

Then load the config and the service provider in `bootstrap/app.php`

```
$app->configure('modules');
$app->register(\Nwidart\Modules\LumenModulesServiceProvider::class)
```

Laravel-modules uses `path.public` which isn't defined by default in Lumen. Register `path.public` before loading the service provider.

```
$app->bind('path.public', function() {
 return __DIR__ . 'public/';
});
```

Questions and Issues

For any bugs, issues and questions for **laravel-modules**, please refer to the github repository, issues tab.

We also have a Discord community. https://discord.gg/hkF7BRvRZK For quick help, ask questions in the appropriate channel.

Configuration

You can publish the package configuration using the following command:

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

In the published configuration file you can configure the following things:

Default namespace

What the default namespace will be when generating modules.

Default: `Modules`

The default namespace is set as Modules this will apply the namespace for all classes the module will use when it's being created and later when generation additional classes.

Overwrite the generated files (stubs)

Overwrite the default generated stubs to be used when generating modules. This can be useful to customise the output of different files.

These stubs set options and paths.

Enabled true or false will enable or disable a module upon creation, the default is false meaning you will have to enable a module manually.

To enable a module edit **module_statuses.json** or run the command:

```
php artisan module:enable ModuleName
```

note the module_statues.json file will be created if it does not exist using this command. The contents of module_statuses.json looks like:

```
{
    "Users": true
}
```

The above would be when there is a single module called Users and is enabled.

Path points to a vendor directly where the default stubs are located, these can be published and modified.

Files set the file locations defaults.

Replacements is a way to do a Find and Replace on generation any matches will be replaced.

Comment out any files below that you do not want to be generated.

For example if you do not intend to use any frontend assets there is no need for CSS/JS files or asset loading files so turn them off:

```
'files' => [
    'routes/web' => 'routes/web.php',
    'routes/api' => 'routes/api.php',
    'views/index' => 'resources/views/index.blade.php',
    'views/master' => 'resources/views/layouts/master.blade.php',
    'scaffold/config' => 'config/config.php',
    'composer' => 'composer.json',
    //'assets/js/app' => 'resources/js/app.js',
    //'assets/sass/app' => 'resources/css/app.css',
    //'vite' => 'vite.config.js',
    //'package' => 'package.json',
],
```

The complete list:

```
'stubs' => [
    'enabled' => false,
    'path' => base_path('vendor/nwidart/laravel-
modules/src/Commands/stubs'),
    'files' => [
        'routes/web' => 'routes/web.php',
        'routes/api' => 'routes/api.php',
        'views/index' => 'resources/views/index.blade.php',
        'views/master' => 'resources/views/layouts/master.blade.php',
        'scaffold/config' => 'config/config.php',
        'composer' => 'composer.json',
        'assets/js/app' => 'resources/assets/js/app.js',
        'assets/sass/app' => 'resources/assets/sass/app.scss',
        'vite' => 'vite.config.js',
        'package' => 'package.json',
    ],
    'replacements' => [
        'routes/web' => ['LOWER_NAME', 'STUDLY_NAME',
'MODULE_NAMESPACE', 'CONTROLLER_NAMESPACE'],
        'routes/api' => ['LOWER_NAME', 'STUDLY_NAME',
'MODULE_NAMESPACE', 'STUDLY_NAME', 'CONTROLLER_NAMESPACE'],
        'vite' => ['LOWER_NAME'],
        'json' => ['LOWER_NAME', 'STUDLY_NAME', 'MODULE_NAMESPACE',
'PROVIDER_NAMESPACE'],
        'views/index' => ['LOWER_NAME'],
        'views/master' => ['LOWER_NAME', 'STUDLY_NAME'],
        'scaffold/config' => ['STUDLY_NAME'],
        'composer' => [
            'LOWER_NAME',
            'STUDLY_NAME',
            'VENDOR',
            'AUTHOR_NAME',
            'AUTHOR_EMAIL',
            'MODULE_NAMESPACE',
            'PROVIDER_NAMESPACE',
        ],
    ],
    'gitkeep' => true,
],
```

Generator Path

These are the files that are generated by default where generate is set to `true`, when false is used that path is not generated.

```
'generator' => [
    // app/
    'channels' => ['path' => 'app/Broadcasting', 'generate' => false],
    'command' => ['path' => 'app/Console', 'generate' => false],
    'emails' => ['path' => 'app/Emails', 'generate' => false],
    'event' => ['path' => 'app/Events', 'generate' => false],
    'jobs' => ['path' => 'app/Jobs', 'generate' => false],
    'listener' => ['path' => 'app/Listeners', 'generate' => false],
    'model' => ['path' => 'app/Models', 'generate' => false],
    'notifications' => ['path' => 'app/Notifications', 'generate' =>
false],
    'observer' => ['path' => 'app/Observers', 'generate' => false],
    'policies' => ['path' => 'app/Policies', 'generate' => false],
    'provider' => ['path' => 'app/Providers', 'generate' => true],
    'repository' => ['path' => 'app/Repositories', 'generate' => false],
    'resource' => ['path' => 'app/Transformers', 'generate' => false],
    'rules' => ['path' => 'app/Rules', 'generate' => false],
    'component-class' => ['path' => 'app/View/Components', 'generate' =>
false],
    // app/HTTP/
    'controller' => ['path' => 'app/Http/Controllers', 'generate' =>
true],
    'filter' => ['path' => 'app/Http/Middleware', 'generate' => false],
    'request' => ['path' => 'app/Http/Requests', 'generate' => false],

    // config/
    'config' => ['path' => 'config', 'generate' => true],

    // database/
    'migration' => ['path' => 'database/migrations', 'generate' =>
false],
    'seeder' => ['path' => 'database/seeders', 'generate' => false],
    'factory' => ['path' => 'database/factories', 'generate' => false],

    // lang/
    'lang' => ['path' => 'lang', 'generate' => false],

    // resource/
```

```
    'assets' => ['path' => 'resources/assets', 'generate' => false],
    'views' => ['path' => 'resources/views', 'generate' => true],
    'component-view' => ['path' => 'resources/views/components',
'generate' => false],

    // routes/
    'routes' => ['path' => 'routes', 'generate' => true],

    // tests/
    'test-unit' => ['path' => 'tests/Unit', 'generate' => false],
    'test-feature' => ['path' => 'tests/Feature', 'generate' => false],
],
```

Don't like Entities for the Models here's where you can change the path to Models instead.

From v11 the config has been changed to use Models instead of Entities, you can change this in your config.

Instead of Entities:

```
'model' => ['path' => 'Entities', 'generate' => true],
```

Use of Models:

```
'model' => ['path' => 'Models', 'generate' => true],
```

Package Commands

As from `v10.0.5` the commands are loaded by an internal `ConsoleServiceProvider`.

The commands are autoloaded internally. You can add any custom commands by using the `->merge()` method.

```
'commands' => ConsoleServiceProvider::defaultCommands()
    ->merge([
        // New commands go here
    ])->toArray(),
```

For existing configs please replace the commands with:

```
'commands' => ConsoleServiceProvider::defaultCommands()
    ->merge([
        // New commands go here
    ])->toArray(),
```

Overwrite the paths

Overwrite the default paths used throughout the package.

Set the path for where to place the Modules folder, where the assets will be published and the location for the migrations.

It's recommend keep the defaults here.

```
'paths' => [
    'modules' => base_path('Modules'),
    'assets' => public_path('modules'),
    'migration' => base_path('database/migrations'),
```

Scan additional folders for modules

By default, modules are loaded from a directory called Modules, in addition to the scan path. Any packages installed for modules can be loaded from here.

```
'scan' => [
    'enabled' => false,
    'paths' => [
        base_path('vendor/*/*'),
    ],
],
```

You can add your own locations for instance say you're building a large application and want to have multiple module folder locations, you can create as many as needed.

```
'scan' => [
    'enabled' => true,
    'paths' => [
        base_path('ModulesCms'),
        base_path('ModulesERP'),
        base_path('ModulesShop'),
    ],
],
```

Remember to set `enabled` to `true` to enable these locations.

Composer file template

When generating a module the composer.json file will contain the author details as set out below, change them as needed.

Take special notice of the vendor, if you plan on extracting modules to packages later it's recommend using your BitBucket/GitHub/GitLab vendor name here.

```
'composer' => [
    'vendor' => 'nwidart',
    'author' => [
        'name' => 'Nicolas Widart',
        'email' => 'n.widart@gmail.com',
    ],
]
```

Caching

If you have many modules it's a good idea to cache this information (like the multiple `module.json` files for example).

Modules can be cached, by default caching is off.

```
'cache' => [
    'enabled' => false,
    'driver' => 'file',
    'key' => 'laravel-modules',
    'lifetime' => 60,
],
```

Registering custom namespace

Decide which custom namespaces need to be registered by the package. If one is set to false, the package won't handle its registration.

Compiling Assets

Vite

When you create a new module it also creates assets for CSS/JS and the `vite.config.js` configuration file.

```
php artisan module:make Blog
```

Change directory to the module:

```
cd Modules/Blog
```

The default `package.json` file includes everything you need to get started. You may install the dependencies it references by running:

```
npm install
```

Or for updating:

```
npm update
```

To learn more about Vite see https://laravel.com/docs/10.x/vite

Next choose between loading vite from a module or main vite.config.js you should not use both.

From version 10.0.3

Vite load from main vite.config.js

When loading assets from all `enabled` modules together follow these steps.

vite.config.js

the default `vite.config.js` needs to be updated. Copy and paste the below configuration.

```
import {defineConfig} from 'vite';
import laravel from 'laravel-vite-plugin';
import collectModuleAssetsPaths from './vite-module-loader.js';

async function getConfig() {
    const paths = [
        'resources/css/app.css',
        'resources/js/app.js',
    ];
    const allPaths = await collectModuleAssetsPaths(paths, 'Modules');

    return defineConfig({
        plugins: [
            laravel({
                input: allPaths,
                refresh: true,
            })
        ]
    });
}

export default getConfig();
```

Inside the paths array set paths to any css/js files from within the resources folder. Next if your modules folder has been changed update the line below to look in the modules folder.

```
const allPaths = await collectModuleAssetsPaths(paths, 'Modules');
```

This config will load all `enabled` modules and read their `vite.config.js` files and compile them into one collection.

This will store the compiled assets into a `public/build` folder.

Module vite.config.js

to load assets from a module edit its `vite.config.php` and replace its contents with:

Update the paths as needed.

```
export const paths = [
    'Modules/Blog/resources/assets/sass/app.scss',
    'Modules/Blog/resources/assets/js/app.js',
];
```

Render

Inside your layout file use the following to load the assets

```
@vite(\Nwidart\Modules\Module::getAssets())
```

This will include the script and css tags automatically.

Running Vite

To compile the assets run

```
npm run build
```

To make use of hot-reloading / watching for changes run

```
npm run dev
```

Vite load from modules

When using layout files within a module to render its assets follow these steps.

Ensure `vite-module-loader.js` has been published to the project root by running:

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

vite.config.js

the default `vite.config.js`:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    build: {
        outDir: '../../public/build-blog',
        emptyOutDir: true,
        manifest: true,
    },
    plugins: [
        laravel({
            publicDirectory: '../../public',
            buildDirectory: 'build-blog',
            input: [
                __dirname + '/resources/assets/sass/app.scss',
                __dirname + '/resources/assets/js/app.js'
            ],
            refresh: true,
        }),
    ],
});
```

This will store `app.scss` and `app.js` from the module into a `public/build-*` folder set by the module name. For a blog module the build folder will be called `build-blog` the build folders contain the paths vite needs to render the compiled files.

Render

Inside the layout file use `module_vite()` to load the assets

```
{{ module_vite('build-blog', 'resources/assets/sass/app.scss') }}
{{ module_vite('build-blog', 'resources/assets/js/app.js') }}
```

Running Vite

To compile the assets run from the module's directory

```
npm run build
```

To make use of hot-reloading / watching for changes run from the module's directory

```
npm run dev
```

Mix

Mix is no longer recommended as Laravel now uses Vite by default.

When you create a new module it also creates assets for CSS/JS and the `webpack.mix.js` configuration file.

```
php artisan module:make Blog
```

Change directory to the module:

```
cd Modules/Blog
```

The default `package.json` file includes everything you need to get started. You may install the dependencies it references by running:

```
npm install
```

Running Mix

Mix is a configuration layer on top of Webpack, so to run your Mix tasks you only need to execute one of the NPM scripts that is included with the default `laravel-modules` `package.json` file

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run production
```

After generating the versioned file, you won't know the exact file name. So, you should use Laravel's global mix function within your views to load the appropriately hashed asset. The mix function will automatically determine the current name of the hashed file:

```
// Modules/Blog/Resources/views/layouts/master.blade.php

<link rel="stylesheet" href="{{ mix('css/blog.css') }}">

<script src="{{ mix('js/blog.js') }}"></script>
```

For more info on Laravel Mix view the documentation here: https://laravel.com/docs/mix

Note: to prevent the main Laravel Mix configuration from overwriting the `public/mix-manifest.json` file:

Install `laravel-mix-merge-manifest`

```
npm install laravel-mix-merge-manifest --save-dev
```

Modify `webpack.mix.js` main file

```
let mix = require('laravel-mix');


/* Allow multiple Laravel Mix applications*/
require('laravel-mix-merge-manifest');
mix.mergeManifest();
/*----------------------------------------*/

mix.js('resources/assets/js/app.js', 'public/js')
   .sass('resources/assets/sass/app.scss', 'public/css');
```

Creating a module

To make modules use the artisan command `php artisan module:make ModuleName` to create a module called Posts:

```
php artisan module:make posts
```

This will create a module in the path `Modules/Posts`

You can create multiple modules in one command by specifying the names separately:

```
php artisan module:make customers contacts users invoices quotes
```

Which would create each module.

Flags

By default, when you create a new module, the command will add some resources like a controller, seed class, service provider, etc. automatically. If you don't want these, you can add `--plain` flag, to generate a plain module.

```
php artisan module:make Blog --plain
```

or

```
php artisan module:make Blog -p
```

Additional flags are as follows:

Generate an api module.

```
php artisan module:make Blog --api
```

Do not enable the module at creation.

```
php artisan module:make Blog --disabled
```

or

```
php artisan module:make Blog -d
```

Folder structure

This is the default structure for new modules, you are free to change this structure as needed by editing the generator paths in the `modules.php` config file.

```
Modules/
  ├── Blog/
      ├──app
          ├── Http/
              ├── Controllers/
                  BlogController.php
          ├── Providers/
              ├── BlogServiceProvider.php
              ├── RouteServiceProvider.php
      ├── config/
          ├──config.php
      ├── database/
          ├── factories/
          ├── migrations/
          ├── seeders/
              ├── BlogDatabaseSeeder.php
       ├── resources/
          ├── assets/
          ├── views/
       ├── routes/
          ├── api.php
          ├── web.php
       ├── tests/
       ├── composer.json
       ├── module.json
       ├── package.json
       ├── vite.config.js
```

Composer.json

Each module has its own composer.json file, this sets the name of the module, its description and author. You normally only need to change this file if you need to change the vendor name or have its own composer dependencies.

For instance say you wanted to install a package into this module:

```
"require": {
    "dcblogdev/laravel-box": "^2.0"
}
```

This would require the package for this module, but it won't be loaded for the main Laravel composer.json file. For that you would have to put the dependency in the Laravel composer.json file. The main reason this exists is for when extracting a module to a package.

Module.json

This file details the name alias and description / options:

```
{
    "name": "Blog",
    "alias": "blog",
    "description": "",
    "keywords": [],
    "priority": 0,
    "providers": [
        "Modules\\Blog\\Providers\\BlogServiceProvider"
    ],
    "files": []
}
```

Modules are loaded in the priority order, change the priority number to have modules booted / seeded in a custom order.

The files option can be used to include files:

```
"files": [
  "start.php"
]
```

Build your own module generator

The default modules are fine to get started but often you'll have your own structure that you'll use from module to module. It's nice to be able to use your structure as a template for all future modules to be created from.

There is an external package for generating Laravel Modules from a template https://github.com/dcblogdev/laravel-module-generator

There's 2 options here. Edit the stub files or create your own base module and custom artisan command.

Let's explore both options

Stub files

By default, the `config/modules.php` file has a stubs path that points to the laravel-modules package vendor:

```
'stubs' => [
    'enabled' => false,
    'path' => base_path() . '/vendor/nwidart/laravel-
modules/src/Commands/stubs',
```

You can change this path to one were you can edit the files. You should never edit files within the vendor folder so instead lets take a copy of the `vendor/nwidart/laravel-modules/src/Commands/stubs` folder to this path `stubs/module`

If you do not have a `stubs` folder then first publish Laravel's stubs:

```
php artisan stub:publish
```

This will create a stubs folder containing all the stub files Laravel used when creating classes. Make a folder called `module` inside `stubs`.

Ensure you've copied the contents of `vendor/nwidart/laravel-modules/src/Commands/stubs` into `stubs/module` now open `config/modules.php` and edit the path for stubs:

```
'stubs' => [
   'enabled' => false,
   'path' => base_path() . '/stubs/module',
```

Now you can edit any of the stubs for instance I like to remove all the docblocks from controllers by default the `controllers.stub` file contains:

```
<?php

namespace $CLASS_NAMESPACE$;

use Illuminate\Contracts\Support\Renderable;
use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class $CLASS$ extends Controller
{
    /**
     * Display a listing of the resource.
     * @return Renderable
     */
    public function index()
    {
        return view('$LOWER_NAME$::index');
    }

    /**
     * Show the form for creating a new resource.
     * @return Renderable
     */
    public function create()
    {
        return view('$LOWER_NAME$::create');
    }
```

```
    /**
     * Store a newly created resource in storage.
     * @param Request $request
     * @return Renderable
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Show the specified resource.
     * @param int $id
     * @return Renderable
     */
    public function show($id)
    {
        return view('$LOWER_NAME$::show');
    }

    /**
     * Show the form for editing the specified resource.
     * @param int $id
     * @return Renderable
     */
    public function edit($id)
    {
        return view('$LOWER_NAME$::edit');
    }

    /**
     * Update the specified resource in storage.
     * @param Request $request
     * @param int $id
     * @return Renderable
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     * @param int $id
     * @return Renderable
```

```
     */
    public function destroy($id)
    {
        //
    }
}
```

I always delete the docblocks so lets edit the stub for this file, open `stubs/module/controllers.stub`

```
<?php

namespace $CLASS_NAMESPACE$;

use Illuminate\Contracts\Support\Renderable;
use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class $CLASS$ extends Controller
{
    public function index()
    {
        return view('$LOWER_NAME$::index');
    }

    public function create()
    {
        return view('$LOWER_NAME$::create');
    }

    public function store(Request $request)
    {
        //
    }

    public function show($id)
    {
        return view('$LOWER_NAME$::show');
    }

    public function edit($id)
    {
        return view('$LOWER_NAME$::edit');
```

```
    }

    public function update(Request $request, $id)
    {
        //
    }

    public function destroy($id)
    {
        //
    }
}
```

Now when you create a module or create a controller:

```
php artisan module:make-controller CustomerController Customers
```

The stubs for controllers.stub will be used.

As you can see it's simple to edit the default files that are created. Now you could modify lots of files to have a structure you want by default but these stubs are used for both creating modules and files. Meaning if you modify the files to your defaults then generating additional classes they will also have the set structure. You may not always want this.

Often I want a starting structure for an entire module and bare-bones boilerplate when generating classes.

Base Module & custom Artisan command

What we're going to do is create an artisan command that will take a copy of a module and rename its files and placeholder words inside the module as a new module.

To generate a new module, you will want to create a module normally first ensure it has everything you want as a base. Keep it simple for instance for a CRUD (Create Read Update and Delete) module you'll want routes to list, add, edit and delete their controller methods views and tests.

Base Module

Once you have a module you're happy as a starting point, you're ready to convert this to a base module. Copy the module to a location. I will be using `stubs/base-module` as the location.

```
stubs/
    base-module
        app/Http
        app/Models
        app/Providers
        config
        database
        resources
        routes
        tests
        composer.json
        module.json
        package.json
        vite.config.js
```

Every reference to a module would need to be changed when making a new module, for instance you have a module called contacts, it has a model called contact everywhere in the module that uses the model would use Contact:: that would need to be changed to the name of the new module when creating a new module.

Instead of manually finding and copying all references to controllers, model, namespaces, views etc you would use placeholders.

Placeholders

These are the placeholders I use:

```
{module_}
```

Module name all lowercase seperate spaces with underscores.

```
{module-}
```

Module name all lowercase seperate spaces with hypens.

```
{Module}
```

Module name in CamelCase.

```
{module}
```

Module name all in lowercase.

```
{Model}
```

Model name in CamelCase.

```
{model}
```

Model name all in lowercase.

These placeholders can then be used throughout the module for example a controller:

```
namespace Modules\{Module}\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;
use Modules\{Module}\Models\{Model};

class {Module}Controller extends Controller
{
    public function index()
    {
        ${module} = {Model}::get();

        return view('{module}::index', compact('{module}'));
    }
}
```

When the placeholders are swapped out look like

```
namespace Modules\Contacts\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;
use Modules\Contacts\Models\Contact;

class {Module}Controller extends Controller
{
    public function index()
    {
        $contacts = Contact::get();

        return view('contacts::index', compact('contacts'));
    }
}
```

This allows for great customisation. You can create any structure you need and later are able to swap out the placeholder for the actual values.

As I've said you would first need to create a base module using these placeholders. In addition, you will want to edit filename such as `Contact.php` for a model to be `Model.php` These would be renamed automatically to the new module name.

Base Module source code

You can find a complete module boilerplate at
https://github.com/modularlaravel/base-module

Let's build a base module here for completeness.

The module will have this structure:

```
base-module
  config
    config.php
  app/Http
    Controllers
        ModuleController.php
    app/Models
        Model.php
    app/Providers
        ModuleServiceProvider.php
        RouteServiceProvider.php
  database
    factories
        ModelFactory.php
    migrations
        create_module_table.php
    seeders
        ModelDatabaseSeeder.php
  resources
    assets
        js
            app.js
        sass
    views
        create.blade.php
        edit.blade.php
        index.blade.php
  routes
    api.php
    web.php
  tests
    Feature
        ModuleTest.php
    Unit
  composer.json
  module.json
  package.json
  vite.config.js
```

Config.php

Will hold the name of the module such as Contacts

```
<?php

return [
    'name' => '{Module}'
];
```

ModelFactory.php

```
<?php
namespace Modules\{Module}\Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Modules\{Module}\Models\{Model};

class {Model}Factory extends Factory
{
    protected $model = {Model}::class;

    public function definition(): array
    {
        return [
            'name' => $this->faker->name()
        ];
    }
}
```

create_model_table.php

```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class Create{Module}Table extends Migration
{
    public function up()
    {
        Schema::create('{module}', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('{module}');
    }
}
```

ModuleDatabaseSeeder.php

```
<?php

namespace Modules\{Module}\Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class {Module}DatabaseSeeder extends Seeder
{
    public function run()
    {
        Model::unguard();

        // $this->call("OthersTableSeeder");
    }
}
```

ModuleController.php

```
<?php

namespace Modules\{Module}\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;
use Modules\{Module}\Models\{Model};

class {Module}Controller extends Controller
{
    public function index()
    {
        ${module} = {Model}::get();

        return view('{module}::index', compact('{module}'));
    }

    public function create()
    {
        return view('{module}::create');
    }

    public function store(Request $request)
```

```
    {
        $request->validate([
            'name' => 'required|string'
        ]);

        {Model}::create([
            'name' => $request->input('name')
        ]);

        return redirect(route('app.{module}.index'));
    }

    public function edit($id)
    {
        ${model} = {Model}::findOrFail($id);

        return view('{module}::edit', compact('{model}'));
    }

    public function update(Request $request, $id)
    {
        $request->validate([
            'name' => 'required|string'
        ]);

        {Model}::findOrFail($id)->update([
            'name' => $request->input('name')
        ]);

        return redirect(route('app.{module}.index'));
    }

    public function destroy($id)
    {
        {Model}::findOrFail($id)->delete();

        return redirect(route('app.{module}.index'));
    }
}
```

Model.php

```php
<?php

namespace Modules\{Module}\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Modules\{Module}\Database\Factories\{Model}Factory;

class {Model} extends Model
{
    use HasFactory;

    protected $fillable = ['name'];
    protected static function newFactory()
    {
        return {Model}Factory::new();
    }
}
```

ModuleServiceProvider.php

```php
<?php

namespace Modules\{Module}\Providers;

use Illuminate\Support\ServiceProvider;

class {Module}ServiceProvider extends ServiceProvider
{
    protected $moduleName = '{Module}';
    protected $moduleNameLower = '{module}';

    public function boot()
    {
        $this->registerTranslations();
        $this->registerConfig();
        $this->registerViews();
        $this->loadMigrationsFrom(module_path($this->moduleName,
'database/migrations'));
    }

    public function register()
```

```
    {
        $this->app->register(RouteServiceProvider::class);
    }

    protected function registerConfig()
    {
        $this->publishes([
            module_path($this->moduleName, 'config/config.php') =>
config_path($this->moduleNameLower . '.php'),
        ], 'config');
        $this->mergeConfigFrom(
            module_path($this->moduleName, 'config/config.php'),
$this->moduleNameLower
        );
    }

    public function registerViews()
    {
        $viewPath = resource_path('views/modules/' .
$this->moduleNameLower);

        $sourcePath = module_path($this->moduleName, 'resources/views');

        $this->publishes([
            $sourcePath => $viewPath
        ], ['views', $this->moduleNameLower . '-module-views']);

$this->loadViewsFrom(array_merge($this->getPublishableViewPaths(),
[$sourcePath]), $this->moduleNameLower);
    }

    public function registerTranslations()
    {
        $langPath = resource_path('lang/modules/' .
$this->moduleNameLower);

        if (is_dir($langPath)) {
            $this->loadJsonTranslationsFrom($langPath,
$this->moduleNameLower);
        } else {
$this->loadJsonTranslationsFrom(module_path($this->moduleName, 'lang'),
$this->moduleNameLower);
        }
    }
```

```
    public function provides()
    {
        return [];
    }

    private function getPublishableViewPaths(): array
    {
        $paths = [];
        foreach (\Config::get('view.paths') as $path) {
            if (is_dir($path . '/modules/' . $this->moduleNameLower)) {
                $paths[] = $path . '/modules/' . $this->moduleNameLower;
            }
        }
        return $paths;
    }
}
```

RouteServiceProvider.php

```php
<?php

namespace Modules\{Module}\Providers;

use Illuminate\Support\Facades\Route;
use Illuminate\Foundation\Support\Providers\RouteServiceProvider as
ServiceProvider;

class RouteServiceProvider extends ServiceProvider
{
    protected $moduleNamespace = '';

    public function boot()
    {
        parent::boot();
    }

    public function map()
    {
        $this->mapApiRoutes();
        $this->mapWebRoutes();
    }

    protected function mapWebRoutes()
    {
        Route::middleware('web')
            ->namespace($this->moduleNamespace)
            ->group(module_path('{Module}', '/Routes/web.php'));
    }

    protected function mapApiRoutes()
    {
        Route::prefix('api')
            ->middleware('api')
            ->namespace($this->moduleNamespace)
            ->group(module_path('{Module}', '/Routes/api.php'));
    }
}
```

create.blade.php

```
@extends('layouts.app')

@section('content')
    <div class="card">
        <h1>Add {Model}</h1>

        <x-form action="{{ route('app.{module}.create') }}">
            <x-form.input name="name" />
            <x-form.button>Submit</x-form.button>
        </x-form>
    </div>
@endsection
```

edit.blade.php

```
@extends('layouts.app')

@section('content')
    <div class="card">
        <h1>Edit {Model}</h1>

        <x-form action="{{ route('app.{module}.update', ${model}->id)
}}" method="patch">
            <x-form.input name="name">{{ ${model}->name }}</x-
form.input>
            <x-form.button>Update</x-form.button>
        </x-form>
    </div>
@endsection
```

index.blade.php

```
@extends('layouts.app')

@section('content')
    <div class="card">
    <h1>{Module}</h1>

    <p><a href="{{ route('app.{module}.create') }}">Add {Model}</a> </p>

    <table>
        <tr>
            <td>Name</td>
            <td>Action</td>
        </tr>
        @foreach(${module} as ${model})
            <tr>
                <td>{{ ${model}->name }}</td>
                <td>
                    <a href="{{ route('app.{module}.edit', ${model}->id)
}}">Edit</a>

                    <a href="#" onclick="event.preventDefault();
document.getElementById('delete-form').submit();">Delete</a>
                    <x-form id="delete-form" method="delete" action="{{
route('app.{module}.delete', ${model}->id) }}" />
                </td>
            </tr>
        @endforeach
    </table>
    </div>
@endsection
```

api.php

```
<?php

use Illuminate\Http\Request;

Route::middleware('auth:api')->get('/{module}', function (Request
$request) {
    return $request->user();
});
```

web.php

```
<?php

use Modules\{Module}\Http\Controllers\{Module}Controller;

Route::middleware('auth')->prefix('app/{module}')->group(function() {
    Route::get('/', [{Module}Controller::class,
'index'])->name('app.{module}.index');
    Route::get('create', [{Module}Controller::class,
'create'])->name('app.{module}.create');
    Route::post('create', [{Module}Controller::class,
'store'])->name('app.{module}.store');
    Route::get('edit/{id}', [{Module}Controller::class,
'edit'])->name('app.{module}.edit');
    Route::patch('edit/{id}', [{Module}Controller::class,
'update'])->name('app.{module}.update');
    Route::delete('delete/{id}', [{Module}Controller::class,
'destroy'])->name('app.{module}.delete');
});
```

ModuleTest.php

```
<?php

use Modules\{Module}\Models\{Model};

uses(Tests\TestCase::class);

test('can see {model} list', function() {
    $this->authenticate();
```

```
    $this->get(route('app.{module}.index'))->assertOk();
});

test('can see {model} create page', function() {
    $this->authenticate();
    $this->get(route('app.{module}.create'))->assertOk();
});

test('can create {model}', function() {
    $this->authenticate();
    $this->post(route('app.{module}.store', [
        'name' => 'Joe'
    ]))->assertRedirect(route('app.{module}.index'));

    $this->assertDatabaseCount('{module}', 1);
});

test('can see {model} edit page', function() {
    $this->authenticate();
    ${model} = {Model}::factory()->create();
    $this->get(route('app.{module}.edit', ${model}->id))->assertOk();
});

test('can update {model}', function() {
    $this->authenticate();
    ${model} = {Model}::factory()->create();
    $this->patch(route('app.{module}.update', ${model}->id), [
        'name' => 'Joe Smith'
    ])->assertRedirect(route('app.{module}.index'));

    $this->assertDatabaseHas('{module}', ['name' => 'Joe Smith']);
});

test('can delete {model}', function() {
    $this->authenticate();
    ${model} = {Model}::factory()->create();
    $this->delete(route('app.{module}.delete',
${model}->id))->assertRedirect(route('app.{module}.index'));

    $this->assertDatabaseCount('{module}', 0);
});
```

composer.json

Ensure you edit the author details here, all future modules will use these details.

```
{
    "name": "dcblogdev/{module}",
    "description": "",
    "authors": [
        {
            "name": "David Carr",
            "email": "dave@dcblog.dev"
        }
    ],
    "extra": {
        "laravel": {
            "providers": [],
            "aliases": {

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

module.json

```
{
    "name": "{Module}",
    "label": "{Module}",
    "alias": "{module}",
    "description": "manage all {module}",
    "keywords": ["{module}"],
    "priority": 0,
    "providers": [
        "Modules\\{Module}\\Providers\\{Module}ServiceProvider"
    ],
    "files": []
}
```

package.json

```
{
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build"
  },
  "devDependencies": {
    "axios": "^1.1.2",
    "laravel-vite-plugin": "^0.7.5",
    "sass": "^1.69.5",
    "postcss": "^8.3.7",
    "vite": "^4.0.0"
  }
}
```

vite.config.js

```
export const paths = [
   'Modules/{Module}/resources/assets/sass/app.scss',
   'Modules/{Module}/resources/assets/js/app.js',
];
```

Artisan make:module command

Now we have a base module and its placeholders in place its time to write an artisan command that will take this as blueprints to create a new module from.

create a new command with this command:

```
php artisan make:command MakeModuleCommand
```

This will generate a new command class inside `app/Console/Commands/MakeModuleCommand.php`

```
<?php

declare(strict_types=1);

namespace App\Console\Commands;

use Illuminate\Console\Command;

class MakeModuleCommand extends Command
{
    protected $signature = 'command:name';
    protected $description = 'Command description';

    public function __construct()
    {
        parent::__construct();
    }

    public function handle()
    {
        return 0;
    }
}
```

We want to call the command using make:module, replace the signature and description:

```
protected $signature = 'make:module';
protected $description = 'Create starter CRUD module';
```

Import Str and Symfony Filesystem classes:

```
use Illuminate\Support\Str;
use Symfony\Component\Filesystem\Filesystem as SymfonyFilesystem;
```

For the Filesystem you would need to install the class via composer:

```
composer require symfony/filesystem
```

Inside the handle method we want the command to ask for a name of the module, we can use `$this->ask` for this.

Ensure the module name is not empty with validation.

When the name of the module is provided, we will try to guess the name of the model and then ask for confirmation if the model name is correct.

If the name is correct confirmation of the module name and model will be printed and ask for final confirmation before moving to the next step.

If confirmation fails the process will restart.

Once the final confirmation has occurred a method called `generate` will be called.

```
public function handle()
{
  $this->container['name'] = ucwords($this->ask('Please enter the name
of the Module'));

  if (strlen($this->container['name']) == 0) {
      $this->error("\nModule name cannot be empty.");
  } else {
      $this->container['model'] =
ucwords(Str::singular($this->container['name']));

      if ($this->confirm("Is '{$this->container['model']}' the correct
name for the Model?", 'yes')) {
          $this->comment('You have provided the following
information:');
          $this->comment('Name:  ' . $this->container['name']);
          $this->comment('Model: ' . $this->container['model']);

          if ($this->confirm('Do you wish to continue?', 'yes')) {
              $this->comment('Success!');
              $this->generate();
          } else {
              return false;
          }

          return true;
      } else {
          $this->handle();
      }
  }
  $this->info('Starter '.$this->container['name'].' module installed
successfully.');
}
```

Now we need to add a generate method, this will need to add a few other methods let's add these first.

Add a method called rename that accepts a path, target and a type. This is used to rename file and save the renamed file into a new `$target` location.

If the `$type` is set to migration then create a timestamp that will be added to the filename and prefixed to the migration file. Otherwise do a direct rename.

```
protected function rename($path, $target, $type = null)
{
    $filesystem = new SymfonyFilesystem;
    if ($filesystem->exists($path)) {
        if ($type == 'migration') {
            $timestamp = date('Y_m_d_his_');
            $target = str_replace("create", $timestamp."create",
$target);
            $filesystem->rename($path, $target, true);
            $this->replaceInFile($target);
        } else {
            $filesystem->rename($path, $target, true);
        }
    }
}
```

Next we want to be able to copy an entire folder and into contents to a new location.

```
protected function copy($path, $target)
{
    $filesystem = new SymfonyFilesystem;
    if ($filesystem->exists($path)) {
        $filesystem->mirror($path, $target);
    }
}
```

The final helper method will be to replace all the placeholders from the files.

Inside this method is where we define the placeholders and their values.

The types defined all the type of placeholders this is case and character sensitive, next each type is looped over. If the key from the type is module_ then all names with spaces will be switched to be underscored separated, likewise for the module- will be hyphened separated for spaces.

Finally, the file's contents will be replaced with the values of the placeholders name of model contents.

```
protected function replaceInFile($path)
{
    $name = $this->container['name'];
    $model = $this->container['model'];
    $types = [
        '{module_}' => null,
        '{module-}' => null,
        '{Module}' => $name,
        '{module}' => strtolower($name),
        '{Model}' => $model,
        '{model}' => strtolower($model)
    ];

    foreach($types as $key => $value) {
        if (file_exists($path)) {

            if ($key == "module_") {
                $parts = preg_split('/(?=[A-Z])/', $name, -1,
PREG_SPLIT_NO_EMPTY);
                $parts = array_map('strtolower', $parts);
                $value = implode('_', $parts);
            }

            if ($key == 'module-') {
                $parts = preg_split('/(?=[A-Z])/', $name, -1,
PREG_SPLIT_NO_EMPTY);
                $parts = array_map('strtolower', $parts);
                $value = implode('-', $parts);
            }

            file_put_contents($path, str_replace($key, $value,
file_get_contents($path)));
        }
    }
}
```

Now we have these helpers methods we can create the generate method.

First we create local variables of name and model for easy referencing. The `$targetPath` variable stores the final module path.

Next we need to copy the base module into `Modules` folder.

The new module at this point is in the modules folder with all the placeholders and temporary file names, now we need to list all files that need the contents examining to place the placeholders.

The last chunk lists all filename that need to be renamed.

```
protected function generate()
{
    $module     = $this->container['name'];
    $model      = $this->container['model'];
    $Module     = $module;
    $module     = strtolower($module);
    $Model      = $model;
    $targetPath = base_path('Modules/'.$Module);

    //copy folders
    $this->copy(base_path('stubs/base-module'), $targetPath);

    //replace contents
    $this->replaceInFile($targetPath.'/config/config.php');
$this->replaceInFile($targetPath.'/database/factories/ModelFactory.php')
;
$this->replaceInFile($targetPath.'/database/migrations/create_module_tab
le.php');
$this->replaceInFile($targetPath.'/database/seeders/ModelDatabaseSeeder.
php');
$this->replaceInFile($targetPath.'/app/Http/Controllers/ModuleController
.php');
    $this->replaceInFile($targetPath.'/app/Models/Model.php');
$this->replaceInFile($targetPath.'/app/Providers/ModuleServiceProvider.p
hp');
$this->replaceInFile($targetPath.'/app/Providers/RouteServiceProvider.ph
p');
$this->replaceInFile($targetPath.'/resources/views/create.blade.php');
    $this->replaceInFile($targetPath.'/resources/views/edit.blade.php');
$this->replaceInFile($targetPath.'/resources/views/index.blade.php');
    $this->replaceInFile($targetPath.'/routes/api.php');
    $this->replaceInFile($targetPath.'/routes/web.php');
    $this->replaceInFile($targetPath.'/tests/Feature/ModuleTest.php');
    $this->replaceInFile($targetPath.'/composer.json');
    $this->replaceInFile($targetPath.'/module.json');
    $this->replaceInFile($targetPath.'/vite.config.js');

    //rename
```

```
    $this->rename($targetPath.'/database/factories/ModelFactory.php',
$targetPath.'/database/factories/'.$Model.'Factory.php');
$this->rename($targetPath.'/database/migrations/create_module_table.php'
, $targetPath.'/database/migrations/create_'.$module.'_table.php',
'migration');
$this->rename($targetPath.'/database/seeders/ModelDatabaseSeeder.php',
$targetPath.'/database/seeders/'.$Module.'DatabaseSeeder.php');
$this->rename($targetPath.'/app/Http/Controllers/ModuleController.php',
$targetPath.'/app/Http/Controllers/'.$Module.'Controller.php');
    $this->rename($targetPath.'/app/Models/Model.php',
$targetPath.'/Models/'.$Model.'.php');
$this->rename($targetPath.'/app/Providers/ModuleServiceProvider.php',
$targetPath.'/app/Providers/'.$Module.'ServiceProvider.php');
    $this->rename($targetPath.'/tests/Feature/ModuleTest.php',
$targetPath.'/tests/Feature/'.$Module.'Test.php');
}
```

Putting it all together:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Str;
use Symfony\Component\Filesystem\Filesystem as SymfonyFilesystem;

class MakeModuleCommand extends Command
{
    protected $signature = 'make:module';
    protected $description = 'Create starter CRUD module';

    public function handle()
    {
        $this->container['name'] = ucwords($this->ask('Please enter the
name of the Module'));

        if (strlen($this->container['name']) == 0) {
            $this->error("\nModule name cannot be empty.");
        } else {
            $this->container['model'] =
ucwords(Str::singular($this->container['name']));
```

```
            if ($this->confirm("Is '{$this->container['model']}' the
correct name for the Model?", 'yes')) {
                $this->comment('You have provided the following
information:');
                $this->comment('Name:  ' . $this->container['name']);
                $this->comment('Model: ' . $this->container['model']);

                if ($this->confirm('Do you wish to continue?', 'yes')) {
                    $this->comment('Success!');
                    $this->generate();
                } else {
                    return false;
                }

                return true;
            } else {
                $this->handle();
            }
        }

        $this->info('Starter '.$this->container['name'].' module
installed successfully.');
    }

    protected function generate()
    {
        $module     = $this->container['name'];
        $model      = $this->container['model'];
        $Module     = $module;
        $module     = strtolower($module);
        $Model      = $model;
        $targetPath = base_path('Modules/'.$Module);

        //copy folders
        $this->copy(base_path('stubs/base-module'), $targetPath);

        //replace contents
        $this->replaceInFile($targetPath.'/config/config.php');
$this->replaceInFile($targetPath.'/database/factories/ModelFactory.php')
;
$this->replaceInFile($targetPath.'/database/migrations/create_module_tab
le.php');
$this->replaceInFile($targetPath.'/database/seeders/ModelDatabaseSeeder.
php');
$this->replaceInFile($targetPath.'/app/Http/Controllers/ModuleController
```

```
.php');
        $this->replaceInFile($targetPath.'/app/Models/Model.php');
$this->replaceInFile($targetPath.'/app/Providers/ModuleServiceProvider.p
hp');
$this->replaceInFile($targetPath.'/app/Providers/RouteServiceProvider.ph
p');
$this->replaceInFile($targetPath.'/resources/views/create.blade.php');
$this->replaceInFile($targetPath.'/resources/views/edit.blade.php');
$this->replaceInFile($targetPath.'/resources/views/index.blade.php');
        $this->replaceInFile($targetPath.'/routes/api.php');
        $this->replaceInFile($targetPath.'/routes/web.php');
$this->replaceInFile($targetPath.'/tests/Feature/ModuleTest.php');
        $this->replaceInFile($targetPath.'/composer.json');
        $this->replaceInFile($targetPath.'/module.json');
        $this->replaceInFile($targetPath.'/vite.config.js');

        //rename
$this->rename($targetPath.'/database/factories/ModelFactory.php',
$targetPath.'/database/factories/'.$Model.'Factory.php');
$this->rename($targetPath.'/database/migrations/create_module_table.php'
, $targetPath.'/database/migrations/create_'.$module.'_table.php',
'migration');
$this->rename($targetPath.'/database/seeders/ModelDatabaseSeeder.php',
$targetPath.'/database/seeders/'.$Module.'DatabaseSeeder.php');
$this->rename($targetPath.'/app/Http/Controllers/ModuleController.php',
$targetPath.'/app/Http/Controllers/'.$Module.'Controller.php');
        $this->rename($targetPath.'/app/Models/Model.php',
$targetPath.'/Models/'.$Model.'.php');
$this->rename($targetPath.'/app/Providers/ModuleServiceProvider.php',
$targetPath.'/app/Providers/'.$Module.'ServiceProvider.php');
        $this->rename($targetPath.'/tests/Feature/ModuleTest.php',
$targetPath.'/tests/Feature/'.$Module.'Test.php');
    }

    protected function rename($path, $target, $type = null)
    {
        $filesystem = new SymfonyFilesystem;
        if ($filesystem->exists($path)) {
            if ($type == 'migration') {
                $timestamp = date('Y_m_d_his_');
                $target = str_replace("create", $timestamp."create",
$target);
                $filesystem->rename($path, $target, true);
                $this->replaceInFile($target);
            } else {
```

```
                $filesystem->rename($path, $target, true);
            }
        }
    }

    protected function copy($path, $target)
    {
        $filesystem = new SymfonyFilesystem;
        if ($filesystem->exists($path)) {
            $filesystem->mirror($path, $target);
        }
    }

    protected function replaceInFile($path)
    {
        $name = $this->container['name'];
        $model = $this->container['model'];
        $types = [
            '{module_}' => null,
            '{module-}' => null,
            '{Module}' => $name,
            '{module}' => strtolower($name),
            '{Model}' => $model,
            '{model}' => strtolower($model)
        ];

        foreach($types as $key => $value) {
            if (file_exists($path)) {

                if ($key == "module_") {
                    $parts = preg_split('/(?=[A-Z])/', $name, -1,
PREG_SPLIT_NO_EMPTY);
                    $parts = array_map('strtolower', $parts);
                    $value = implode('_', $parts);
                }

                if ($key == 'module-') {
                    $parts = preg_split('/(?=[A-Z])/', $name, -1,
PREG_SPLIT_NO_EMPTY);
                    $parts = array_map('strtolower', $parts);
                    $value = implode('-', $parts);
                }

                file_put_contents($path, str_replace($key, $value,
file_get_contents($path)));
```

```
                }
            }
        }
    }
```

This seems like a lot of work but this gives you complete control over what a module will contain when generated. For simple CRUD modules you can build an application incredibly quickly with this approach.

Helpers

Module path function

Get the path to the given module.

```
$path = module_path('Blog');
```

Returns absolute path of project ending with /Modules/Blog

`module_path` can take a string as a second param, which tacks on to the end of the path:

```
$path = module_path('Blog', 'Http/controllers/BlogController.php');
```

Returns absolute path of project ending with
/Modules/Blog/Http/controllers/BlogController.php

Artisan commands

Useful Tip:

You can use the following commands with the `--help` suffix to find its arguments and options.

Note all the following commands use "Blog" as example module name, and example class/file names

Utility commands

module:make

Generate a new module.

```
php artisan module:make Blog
```

module:make

Generate multiple modules at once.

```
php artisan module:make Blog User Auth
```

module:use

Use a given module. This allows you to not specify the module name on other commands requiring the module name as an argument.

```
php artisan module:use Blog
```

module:unuse

This unsets the specified module that was set with the `module:use` command.

```
php artisan module:unuse
```

module:list

List all available modules.

```
php artisan module:list
```

module:show-model

Provides a convenient overview of all the model's attributes and relations:

```
php artisan module:show-model Blog
```

module:migrate

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

```
php artisan module:migrate Blog
```

module:migrate-rollback

Rollback the given module, or without an argument, rollback all modules.

```
php artisan module:migrate-rollback Blog
```

From version 10.1

To only back a specific migration use the option `--subpath`

```
php artisan module:migrate-rollback --
subpath="2023_10_17_101427_create_posts_table.php" Blog
```

module:migrate-refresh

Refresh the migration for the given module, or without a specified module refresh all modules migrations.

```
php artisan module:migrate-refresh Blog
```

module:migrate-reset Blog

Reset the migration for the given module, or without a specified module reset all modules migrations.

```
php artisan module:migrate-reset Blog
```

module:seed

Seed the given module, or without an argument, seed all modules

```
php artisan module:seed Blog
```

module:publish-migration

Publish the migration files for the given module, or without an argument publish all modules migrations.

```
php artisan module:publish-migration Blog
```

module:publish-config

Publish the given module configuration files, or without an argument publish all modules configuration files.

```
php artisan module:publish-config Blog
```

module:publish-translation

Publish the translation files for the given module, or without a specified module publish all modules migrations.

```
php artisan module:publish-translation Blog
```

module:lang

Check missing language keys in the specified module.

```
php artisan module:lang Blog
```

module:enable

Enable the given module.

```
php artisan module:enable Blog
```

module:disable

Disable the given module.

```
php artisan module:disable Blog
```

module:update

Update the given module.

```
php artisan module:update Blog
```

Generator commands

module:make-command

Generate the given console command for the specified module.

```
php artisan module:make-command CreatePostCommand Blog
```

module:make-migration

Generate a migration for specified module.

```
php artisan module:make-migration create_posts_table Blog
```

module:make-seed

Generate the given seed name for the specified module.

```
php artisan module:make-seed seed_fake_blog_posts Blog
```

module:make-controller

Generate a controller for the specified module.

```
php artisan module:make-controller PostsController Blog
```

Optional options:

- `--plain`,`-p` : create a plain controller
- `--api` : create a resouce controller

module:make-model

Generate the given model for the specified module.

```
php artisan module:make-model Post Blog
```

Optional options:

- `--fillable=field1,field2`: set the fillable fields on the generated model
- `--migration`, `-m`: create the migration file for the given model
- `--request`, `-r`: create the request file for the given model
- `--seed`, `-s`: create the seeder file for the given model
- `--controller`, `-c`: create the controller file for the given model
- `-mcrs`: create migration, controller, request and seeder files all together for the given model

module:make-provider

Generate the given service provider name for the specified module.

```
php artisan module:make-provider BlogServiceProvider Blog
```

module:make-middleware

Generate the given middleware name for the specified module.

```
php artisan module:make-middleware CanReadPostsMiddleware Blog
```

module:make-mail

Generate the given mail class for the specified module.

```
php artisan module:make-mail SendWeeklyPostsEmail Blog
```

module:make-notification

Generate the given notification class name for the specified module.

```
php artisan module:make-notification NotifyAdminOfNewComment Blog
```

module:make-listener

Generate the given listener for the specified module. Optionally you can specify which event class it should listen to. It also accepts a `--queued` flag allowed queued event listeners.

```
php artisan module:make-listener NotifyUsersOfANewPost Blog
php artisan module:make-listener NotifyUsersOfANewPost Blog --
event=PostWasCreated
php artisan module:make-listener NotifyUsersOfANewPost Blog --
event=PostWasCreated --queued
```

module:make-request

Generate the given request for the specified module.

```
php artisan module:make-request CreatePostRequest Blog
```

module:make-event

Generate the given event for the specified module.

```
php artisan module:make-event BlogPostWasUpdated Blog
```

module:make-job

Generate the given job for the specified module.

```
php artisan module:make-job JobName Blog

php artisan module:make-job JobName Blog --sync # A synchronous job
class
```

module:route-provider

Generate the given route service provider for the specified module.

```
php artisan module:route-provider Blog
```

module:make-factory

Generate the given database factory for the specified module.

```
php artisan module:make-factory ModelName Blog
```

module:make-policy

Generate the given policy class for the specified module.

The `Policies` is not generated by default when creating a new module. Change the value of `paths.generator.policies` in `modules.php` to your desired location.

```
php artisan module:make-policy PolicyName Blog
```

module:make-rule

Generate the given validation rule class for the specified module.

The `Rules` folder is not generated by default when creating a new module. Change the value of `paths.generator.rules` in `modules.php` to your desired location.

```
php artisan module:make-rule ValidationRule Blog
```

module:make-resource

Generate the given resource class for the specified module. It can have an optional `--collection` argument to generate a resource collection.

The `Transformers` folder is not generated by default when creating a new module. Change the value of `paths.generator.resource` in `modules.php` to your desired location.

```
php artisan module:make-resource PostResource Blog
php artisan module:make-resource PostResource Blog --collection
```

module:make-test

Generate the given test class for the specified module.

```
php artisan module:make-test EloquentPostRepositoryTest Blog
```

v11.0.1 added: make:view

module:make-view

Generate the given view for the specified module.

```
php artisan module:make-view index Blog
```

Facade methods

Import facade:

```
use Nwidart\Modules\Facades\Module;
```

Get all modules.

```
Module::all();
```

Get all cached modules.

```
Module::getCached()
```

Get ordered modules. The modules will be ordered by the `priority` key in `module.json` file.

```
Module::getOrdered();
```

Get scanned modules.

```
Module::scan();
```

Find a specific module.

```
Module::find('name');
// OR
Module::get('name');
```

Find a module, if there is one, return the `Module` instance, otherwise throw `Nwidart\Modules\Exeptions\ModuleNotFoundException`.

```
Module::findOrFail('module-name');
```

Get scanned paths.

```
Module::getScanPaths();
```

Get all modules as a collection instance.

```
Module::toCollection();
```

Get modules by the status. 1 for active and 0 for inactive.

```
Module::getByStatus(1);
```

Check the specified module. If it exists, will return `true`, otherwise `false`.

```
Module::has('blog');
```

Get all enabled modules.

```
Module::allEnabled();
```

Check if mobile is enabled

```
Module::isEnabled('ModuleName');
```

Get all disabled modules.

```
Module::allDisabled();
```

Check if mobile is disabled

```
Module::isDisabled('ModuleName');
```

Enable module

```
Module::enable('ModuleName');
```

Disable module

```
Module::disable('ModuleName');
```

Delete module

```
Module::delete('ModuleName');
```

Update dependencies for the specified module.

```
Module::update('hello');
```

Install the specified module by given module name.

```
Module::install('nwidart/hello');
```

Get count of all modules.

```
Module::count();
```

Get module root path.

```
Module::getPath();
```

Get the module's `app` path. If the module doesn't have an `app` folder, the result is the same as `getPath` method.

```
Module::getAppPath();
```

Register the modules.

```
Module::register();
```

Boot all available modules.

```
Module::boot();
```

Get all enabled modules as collection instance.

```
Module::collections();
```

Get module path from the specified module.

```
Module::getModulePath('name');
```

Get assets path from the specified module.

```
Module::assetPath('name');
```

Get config value from this package.

```
Module::config('composer.vendor');
```

Get used storage path.

```
Module::getUsedStoragePath();
```

Get used module for cli session.

```
Module::getUsedNow();
// OR
Module::getUsed();
```

Set used module for cli session.

```
Module::setUsed('name');
```

Get modules's assets path.

```
Module::getAssetsPath();
```

Get asset url from specific module.

```
Module::asset('blog:img/logo.img');
```

Add a macro to the module repository.

```
Module::macro('hello', function() {
    echo "I'm a macro";
});
```

Call a macro from the module repository.

```
Module::hello();
```

Get all required modules of a module

```
Module::getRequirements('module name');
```

Module Console Commands

Module names should not be typed in UPPERCASE but CamelCase. ie instead of SERIALCODE type SerialCode

Your module may contain console commands. You can make these commands manually, or generate them with the following command:

```
php artisan module:make-command CommandName ModuleName
```

ie

```
php artisan moddule:make-command ImportTicketsCommand Tickets
```

This will create an `ImportTicketsCommand` command inside the `Tickets` module located at `Modules/Tickets/Console/ImportTicketsCommand`.

Please refer to the laravel documentation on artisan commands to learn all about them.

Registering commands

In order to use custom module commands, they first need to be registered inside the module service providers `boot` method:

```
$this->commands([
    \Modules\Tickets\Console\ImportTicketsCommand::class,
]);
```

You can now access your command via `php artisan` in the console.

Schedule commands

To use the Artisan's scheduler

Import `Schedule`

```
use Illuminate\Console\Scheduling\Schedule;
```

Now inside the `boot` method called `app->booted` inside the closure make an instance of schedule then register the command to be called and its frequency.

```
$this->app->booted(function () {
    $schedule = $this->app->make(Schedule::class);
    $schedule->command('tickets:import')->everyMinute();
});
```

Module Methods

Get an entity from a specific module.

```
$module = Module::find('blog');
```

Get module name.

```
$module->getName();
```

Get module name in lowercase.

```
$module->getLowerName();
```

Get module name in studlycase.

```
$module->getStudlyName();
```

Get module path.

```
$module->getPath();
```

Get extra path.

```
$module->getExtraPath('Assets');
```

Disable the specified module.

```
$module->disable();
```

Enable the specified module.

```
$module->enable();
```

Check if enabled:

```
$module->isEnabled();
```

Check if disabled:

```
$module->isDisabled);
```

Check if the status it true or false by passing true or false:

```
$module->IsStatus(false); //returns true if active false if is not
active
```

You can also do a call in one go:

```
Module::find('Posts')->isEnabled();
```

Only run the one liner if you've first checked the module exists otherwise you will get an error.

Delete the specified module.

```
$module->delete();
```

Get an array of module requirements. Note: these should be aliases of the module.

```
$module->getRequires();
```

Languages

Array translation strings

```
$langPath = resource_path('lang/modules/'.$this->moduleNameLower);

if (is_dir($langPath)) {
    $this->loadTranslationsFrom($langPath, $this->moduleNameLower);
    $this->loadJsonTranslationsFrom($langPath);
} else {
    $this->loadTranslationsFrom(module_path($this->moduleName, 'lang'),
$this->moduleNameLower);
    $this->loadJsonTranslationsFrom(module_path($this->moduleName,
'lang'));
}
```

use in blade `{{ __(blog::pages.about) }}` will be searched in:
`/lang/modules/en/pages.php` `/Modules/Blog/lang/en/pages.php`

JSON translation strings

To use the translation strings as keys you will need to place a JSON lang file in `Modules/ModuleName/resources/lang` ie fr.json for French.

The file can contain your language variants:

```
{
  "Hello World": "Bonjour le monde"
}
```

By default, modules look for `lang/en/messages.php` file to tell the module to use JSON translations instead.

Find

```
if (is_dir($langPath)) {
    $this->loadTranslationsFrom($langPath, $this->moduleNameLower);
} else {
    $this->loadTranslationsFrom(module_path($this->moduleName,
'Resources/lang'), $this->moduleNameLower);
}
```

And swap the `loadTranslationsFrom` to `loadJsonTranslationsFrom`

Now you can use JSON translations in your controllers and views:

```
<div>
   <lable for="name">{{ __('Name') }}</lable>
   <input type="text" name="name" id="name" value="{{ old('name') }}">
</div>
<div>
   <lable for="subject">{{ __('Subject') }}</lable>
   <input type="text" name="subject" id="subject" value="{{
old('subject') }}">
</div>
```

The French fr.json file would contain:

```
{
  "Name": "Nom",
  "Subject": "Sujette",
}
```

Languages

`phpunit.xml` is the configuration file for phpunit by default this file is configured to run tests only in the `/tests/Feature` and `/tests/Unit`

In order for your modules to be tested their paths need adding to this file, manually adding entries for each module is not desirable so instead use a wildcard include to include the modules dynamically:

```
<testsuite name="Modules">
    <directory suffix="Test.php">./Modules/*/tests/Feature</directory>
    <directory suffix="Test.php">./Modules/*/tests/Unit</directory>
</testsuite>
```

Also ensure you've got the database connection set to sqlite and to use an in-memory database:

```
<server name="DB_CONNECTION" value="sqlite"/>
<server name="DB_DATABASE" value=":memory:"/>
```

Example phpunit.xml

The file would look like this

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="./vendor/phpunit/phpunit/phpunit.xsd"
         bootstrap="vendor/autoload.php"
         colors="true"
>
    <testsuites>
        <testsuite name="Unit">
            <directory suffix="Test.php">./tests/Unit</directory>
        </testsuite>
        <testsuite name="Feature">
            <directory suffix="Test.php">./tests/Feature</directory>
        </testsuite>
        <testsuite name="Modules">
            <directory
suffix="Test.php">./Modules/*/tests/Feature</directory>
            <directory
suffix="Test.php">./Modules/*/tests/Unit</directory>
        </testsuite>
    </testsuites>
    <coverage processUncoveredFiles="true">
        <include>
            <directory suffix=".php">./app</directory>
        </include>
    </coverage>
    <php>
        <server name="APP_ENV" value="testing"/>
        <server name="BCRYPT_ROUNDS" value="4"/>
        <server name="CACHE_DRIVER" value="array"/>
        <server name="DB_CONNECTION" value="sqlite"/>
        <server name="DB_DATABASE" value=":memory:"/>
        <server name="MAIL_MAILER" value="array"/>
        <server name="QUEUE_CONNECTION" value="sync"/>
        <server name="SESSION_DRIVER" value="array"/>
        <server name="TELESCOPE_ENABLED" value="false"/>
    </php>
</phpunit>
```

Now when phpunit is run tests from modules and the global tests folder will run.

Run test for a single module

When you run phpunit:

```
vendor/bin/phpunit
```

All test will run.

Run single test

If you want to only run a single test method, class or module you can use the filter flag:

```
vendor/bin/phpunit --filter 'contacts'
```

This would run all tests inside the Contacts module.

Run a single test method:

```
vendor/bin/phpunit --filter 'can_delete_contact'
```

This would match a test

```
/** @test */
public function can_delete_contact(): void
{
    $this->authenticate();
    $contact = Contact::factory()->create();
    $this->delete(route('app.contacts.delete',
$contact->id))->assertRedirect(route('app.contacts.index'));

    $this->assertDatabaseCount('contacts', 0);
}
```

Test a full test file:

```
vendor/bin/phpunit --filter 'ContactTest'
```

Use PestPHP

Using PestPHP https://pestphp.com.

Often you will want to run Pest testcase class, import it at the top of your file.

```
uses(Tests\TestCase::class);
```

When using Pest you don't need classes the following would be a valid file:

```
<?php

use Modules\Contacts\Models\Contact;

uses(Tests\TestCase::class);

test('can see contact list', function() {
    $this->authenticate();
   $this->get(route('app.contacts.index'))->assertOk();
});

test('can delete contact', function() {
    $this->authenticate();
    $contact = Contact::factory()->create();
    $this->delete(route('app.contacts.delete',
$contact->id))->assertRedirect(route('app.contacts.index'));

    $this->assertDatabaseCount('contacts', 0);
});
```

Run test suite:

```
vendor/bin/pest
```

If you want to only run a single test method, class or module you can use the filter flag:

```
vendor/bin/pest --filter 'contacts'
```

This would match a test

```
test('can_delete_contact', function() {
    $this->authenticate();
    $contact = Contact::factory()->create();
    $this->delete(route('app.contacts.delete',
$contact->id))->assertRedirect(route('app.contacts.index'));

    $this->assertDatabaseCount('contacts', 0);
});
```

Publishing Modules

After creating a module and you are sure your module will be used by other developers. You can push your module to github, gitlab or bitbucket and after that you can submit your module to the packagist website.

You can follow this step to publish your module.

1. Create A Module.
2. Make sure that you mentioned the `type` of the module in the `composer.json` as `laravel-module`
3. Push the module to github, bitbucket, gitlab etc. Make sure the repository name follows the convention then it will be moved to the right directory automatically. The repo name should be like `<namespace>/<name>-module`, a `-module` at the end. Example: `https://github.com/nWidart/article-module`. This module will be installed in `Module/Article` directory.
4. Submit your module to the packagist website. Submit to packagist is very easy, just give your github repository, click submit and you done.

Have modules be installed in the `Modules/` folder

Published modules can be installed like other composer packages. In any Laravel project install the nwidart/laravel-modules package by following the instruction and then you can install your own modules. One extra step you need to take to install the module into the `Modules` directory of the project.

The extra step is to install an additional composer plugin, joshbrw/laravel-module-installer which will move the module files automatically. If you need to install the modules other than the `Modules` directory then add the following in your module composer.json.

```
"extra": {
    "module-dir": "Custom"
}
```

After installing the composer plugin onces, now to install the module you have to use the composer command as like other regular packages,

```
composer require nwidart/article-module
```

A closer look

Important

A module will need to be in its own directly in order to publish only the single module to GIT. Work on the module in a project, when its ready to be converted into a package extract the module to its own folder and rename the module to be all lowercase and ensure it has -module on the end. For example, a module called `Quotes` would be renamed to `quotes-module`.

GIT

Now it's time to set up GIT.

initialise git by typing in terminal:

```
git init
```

This creates a new git instance.

Then add all your files and commit them:

```
git add .
git commit -m 'first commit'
```

From this point onwards any changes you make can be committed to GIT.

Setup repo on BitBucket

A repository needs to be created in Bitbucket click the + button in the sidebar and click repository. direct URL is https://bitbucket.org/repo/create

Enter a project and repository name. Name the repository in the format of `quotes-module`.

For the branch enter `main` and for gitignore select No, you don’t want any files being generated.

Add the remote origin in GIT, go back to your package and in terminal enter replace quotes with the name of the module. Also change the username to match your own.

This will connect Bitbucket to the package.

```
git remote add origin git@bitbucket.org:username/quotes-module.git
```

To upload the package type:

```
git push -u origin main
```

This is only required for the first time, after then a `git push` can be used to push up changes.

Now the module has been pushed to Bitbucket as a package and can be installed into other projects.

Setup repo on GitHub

A repository needs to be created in GitHub by going to https://github.com/new

Enter a repository name. Name the repository in the format of `quotes-module`

Don't check any of the boxes for Initialise this repository with, you don’t want any files being generated.

Click Create Repository.

Now you're ready to upload your local module.

Add the remote origin in GIT, go back to your package and in terminal enter replace quotes with the name of the module. Also change the username to match your own.

This will connect GitHub to the package.

```
git remote add origin git@github.com:username/quotes-module.git
```

To upload the package type:

```
git push -u origin main
```

This is only required for the first time, after then a `git push` can be used to push up changes.

Now the module has been pushed to Bitbucket as a package and can be installed into other projects.

Install a module package (for private packages)

To install a module package open `composer.json`

In the `require` section type: (replace `moduleName` with the name of the package)

```
"vendorname/moduleName-module": "@dev"
```

Next, add a repositories section again replace `moduleName` with the name of the package.

```
"repositories": [
  {
      "type": "vcs",
      "url": "git@bitbucket.org:vendorname/moduleName-module.git"
  }
]
```

Now you can run `composer update` to install the package.

To have modules installed into the Modules directory install this package https://github.com/joshbrw/laravel-module-installer

which will move the module files automatically. If you need to install the modules other than the `Modules` directory then add the following in your module composer.json.

```
"extra": {
    "module-dir": "Custom"
}
```

Setup Module

To activate the module type:

```
php artisan module:enable moduleName
```

Now the module is ready to be migrated and seeded:

```
php atisan module:migrate moduleName
php artisan module:seed moduleName
```

Module updates

You are free to modify the module once installed, these changes are project-specific but if they are generic enough can be added to the package with the normal pull request flow the same as with projects.

If a package has been updated and you need to update the local copy you can do so by doing `composer update` be warned if any of the files have been modified you will get a warning:

```
 Syncing toppackages/suppliers-module (dev-master 852c6b7) into cache
    toppackages/suppliers-module has modified files:
    M Resources/views/livewire/edit.blade.php
    Discard changes [y,n,v,d,s,?]?
```

Type a letter to continue:

y - discard changes and apply the update n - abort the update and let you manually clean things up v - view modified files d - view local modifications (diff) s - stash changes and try to reapply them after the update

Uninstall Packages

If you uninstall a package from `composer.json` the module **WILL** be removed from the modules directory.

Registering Module Events

Your module may contain events and event listeners. You can create these classes manually, or with the following artisan commands:

Generate an event:

```
php artisan module:make-event BlogPostWasUpdatedEvent Blog
```

Generate an event listener:

```
php artisan module:make-listener NotifyAdminOfNewPostListener Blog
```

Once those are create you need to register them in laravel. This can be done in 2 ways:

Manually registering events

Register events manually in your module service provider register method:

```
$this->app['events']->listen(BlogPostWasUpdatedEvent::class,
NotifyAdminOfNewPostListener::class);
```

Creating an EventServiceProvider

Once you have multiple events, you might find it easier to have all events and their listeners in a dedicated service provider. This is what the EventServiceProvider is for.

Create a new class called for instance `EventServiceProvider` in the `Modules/Blog/Providers` folder (Blog being an example name).

This class needs to look like this:

```
<?php

namespace Modules\Blog\App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [];
}
```

Don't forget to load this service provider in the `register` method of the `ModuleServiceProvider` class.

```
$this->app->register(EventServiceProvider::class);
```

This is now like the regular EventServiceProvider in the `app/` namespace. In our example the `listen` property will look like this:

```
<?php

namespace Modules\Blog\App\Events;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        BlogPostWasUpdatedEvent::class => [
            NotifyAdminOfNewPostListener::class,
        ],
    ];
}
```

Livewire

Livewire components

Laravel modules has no make-livewire command. You can manually write the Livewire classes and views.

In order to use Livewire components you'll need to register them in the module service provider, Inside the boot method define the Livewire component and its path.

`component()` takes 2 parameters:

1. The path to the blade views prefixed by the module name
2. The path to the Livewire class

```
Livewire::component('contacts::add', Add::class);
```

To render a Livewire component in a blade view:

```
<livewire:contacts::add />
```

For multi-word class name use hyphens for example a Livewire component called ContactForm:

```
Livewire::component('contacts::contact-form', ContactForm::class);
```

Then render it in a view:

```
<livewire:contacts::contact-form />
```

Livewire Module Package

Whist this will work, it would be far better if you could generate livewire components for modules, good news, there's a package for that by **Mehediul Hassan Miton** called laravel-modules-livewire. https://github.com/mhmiton/laravel-modules-livewire

This package automatically registered livewire components, no longer will you have to register all your Livewire components inside service providers.

Better yet you can generate new Livewire components using a `make-livewire` command.

Installation:

Install through composer:

```
composer require mhmiton/laravel-modules-livewire
```

Config

Publish the package's configuration file:

```
php artisan vendor:publish --
provider="Mhmiton\LaravelModulesLivewire\LaravelModulesLivewireServicePr
ovider"
```

Making Components:

Command Signature:

```
php artisan module:make-livewire <Component> <Module> --view= --force --
inline --custom
```

Example:

```
php artisan module:make-livewire Pages/AboutPage Core
php artisan module:make-livewire Pages\\AboutPage Core
php artisan module:make-livewire pages.about-page Core
```

Force create component if the class already exists:

```
php artisan module:make-livewire Pages/AboutPage Core --force
```

Output:

```
COMPONENT CREATED

CLASS: Modules/Core/Http/Livewire/Pages/AboutPage.php
VIEW:  Modules/Core/Resources/views/livewire/pages/about-page.blade.php
TAG: <livewire:core::pages.about-page />
```

Inline Component:

```
php artisan module:make-livewire Core Pages/AboutPage --inline
```

Output:

```
COMPONENT CREATED

CLASS: Modules/Core/Http/Livewire/Pages/AboutPage.php
TAG: <livewire:core::pages.about-page />
```

Extra Option (--view):

You're able to set a custom view path for component with (--view) option.

Example:

```
php artisan module:make-livewire Pages/AboutPage Core --view=pages/about
php artisan module:make-livewire Pages/AboutPage Core --view=pages.about
```

Output:

```
COMPONENT CREATED

CLASS: Modules/Core/Http/Livewire/Pages/AboutPage.php
VIEW:  Modules/Core/Resources/views/livewire/pages/about.blade.php
TAG: <livewire:core::pages.about-page />
```

Rendering Components:

```
<livewire:{module-lower-name}::component-class-kebab-case />
```

Example:

```
<livewire:core::pages.about-page />
```

Custom Module:

To create components for the custom module, add custom modules in the config file.

The config file is located at `config/modules-livewire.php` after publishing the config file.

Remove comment for these lines & add your custom modules.

```
    /*
    |---------------------------------------------------------------------
-------
    | Custom modules setup
    |---------------------------------------------------------------------
-------
    |
    */

    // 'custom_modules' => [
    //     'Chat' => [
    //         'path' => base_path('libraries/Chat'),
    //         'module_namespace' => 'Libraries\\Chat',
    //         // 'namespace' => 'Http\\Livewire',
    //         // 'view' => 'Resources/views/livewire',
    //         // 'name_lower' => 'chat',
    //     ],
    // ],
```

Custom module config details

path: Add module full path (required).
module_namespace: Add module namespace (required).
namespace: By default using `config('modules-livewire.namespace')` value. You can set a different value for the specific module.
view: By default using `config('modules-livewire.view')` value. You can set a different value for the specific module.
name_lower: By default using module name to lowercase. If you set a custom name, module components will be registered by custom name.

Livewire full-page components

With Livewire you can render components as full pages, instead of using controllers you would only use the Livewire class. Let's go over how to set up a full page component.

Create a Livewire component called Feedback inside a contacts module.

```
php artisan module:make-livewire Feedback Contacts
```

This would output:

```
 COMPONENT CREATED

CLASS: Modules/Contacts/Http/Livewire/Feedback.php
VIEW:  Modules/Contacts/Resources/views/livewire/feedback.blade.php
TAG: <livewire:contacts::feedback />
```

Now setup the route open the module `web.php` file

```
<?php

use Modules\Contacts\Http\Livewire\Feedback;

Route::middleware(['web',
'auth'])->prefix('app/contacts')->group(function() {
    Route::get('feedback',
Feedback::class)->name('app.contacts.feedback');
});
```

Create a route for app/contacts/feedback. Routing directly to `Feedback::class` we do not specify a method only the class.

The Livewire class is a standard livewire class:

```
<?php

namespace Modules\Contacts\Http\Livewire;

use Livewire\Component;

class Feedback extends Component
{
    public function render()
    {
        return view('contacts::livewire.feedback');
    }
}
```

the view feedback.blade.php:

```
<div>
    <h3>The <code>Feedback</code> livewire component is loaded from the
<code>Contacts</code> module.</h3>
</div>
```

At this point there's not much different than embedded Livewire components except you don't have to render the component into a parent view file as its rendered directly.

By default, Livewire will use a layout from `layouts/app` If you want to use a layout from a module you can use `->layout()` on the render method.

```
public function render()
{
    return
view('contacts::livewire.feedback')->layout('contacts::layouts.app');
}
```

Using Spatie permissions package with modules

Spatie provide a powerful roles and permissions package for Laravel. it's a great way to manage complete roles each with their own permissions.

Consult their docs for complete details https://spatie.be/docs/laravel-permission/v5/

Installation

Install the package with composer:

```
composer require spatie/laravel-permission
```

Publish the migrations and config file:

```
php artisan vendor:publish --
provider="Spatie\Permission\PermissionServiceProvider"
```

This shows these file have been published:

```
Copied File [/vendor/spatie/laravel-permission/config/permission.php] To
[/config/permission.php]
Copied File [/vendor/spatie/laravel-
permission/database/migrations/create_permission_tables.php.stub] To
[/database/migrations/2021_12_22_111730_create_permission_tables.php]
Publishing complete.
```

At this point the docs recommending migrating the database, we're not ready yet. We need to alter a migration to include modules but first we want to move all roles and permissions related files to a new module called `Roles`.

Create a roles module:

```
php artisan module:make Roles
```

Now move the `create_permission_tables.php` file from `database/migrations` to the new Roles module `Modules/Roles/database/migrations`

Open the `create_permission_tables.php` file add a modules enter to the permissions schema:

```
Schema::create($tableNames['permissions'], function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->string('name');       // For MySQL 8.0 use string('name',
125);
    $table->string('module');     // For MySQL 8.0 use string('module',
125);
    $table->string('guard_name'); // For MySQL 8.0 use
string('guard_name', 125);
    $table->timestamps();

    $table->unique(['name', 'guard_name']);
});
```

When creating permission a module key should be used in order to group permissions to their modules.

For example:

This would come from the seeder classes from each module, this example is from an admin module

Create permission to view dashboard, the module to an admin module and use the web guard.

```
use Spatie\Permission\Models\Permission;

Permission::firstOrCreate(['name' => 'View Dashboard', 'module' =>
'Admin', 'guard_name' => 'web']);
```

Then to check the permissions the usage is the same as the package documentation, you do not check the module setting here. The Module setting is used only to group the permissions into an UI for managing the permissions by their modules.

```
auth()->user()->hasPermissionTo('View Dashboard')
```

Seed a user with a role

Often you'll want to create a default user with an application and have them set as an Admin, to do this you can create a user inside a seed class. For this I use `Modules/Roles/Database/Seeders/RolesDatabaseSeeder.php` inside the `run` method.

Roles should already exist before trying to assign them:

```
use Spatie\Permission\Models\Role;

Role::firstOrCreate(['name' => 'Admin']);
Role::firstOrCreate(['name' => 'User']);
```

Using firstOrCreate means the seeder can run multiple times, only one user would ever be created this way.

Once the user is created a role of Admin is assigned to the user.

```
public function run()
{
    app()['cache']->forget('spatie.permission.cache');

    $admin = User::firstOrCreate([
        'name' => 'Joe',
        'slug' => 'Bloggs',
        'email' => 'j.bloggs@domain.com'
    ],
    [
        'password' => bcrypt('a-random-password')
    ]);

    $admin->assignRole('Admin');
}
```

Assign all permissions to a role

If you need to assign all permissions to a role, collect an array of the permission ids:

```
$permissions = Permission::all()->pluck('id')->toArray();
```

Then select a role and sync the permissions array:

```
$role = Role::find(1);
$role->syncPermissions($permissions);
```

To group permissions by their roles

Get a specific role.

Get all modules from permissions, use distinct to remove duplicate module names.

```
$role             = Role::findOrFail($id);
$permissionGroups =
Permission::orderBy('module')->get()->groupBy('module');
```

Then in a view loop over the modules array,

```
@foreach($permissionGroups as $module => $permissions)
```

Inside each loop extract their permissions, to display the module name:

```
Str::camel($module)
```

To show all permissions for the module:

```
@foreach ($permissions as $perm)
```

Then using the `$perm` to show the permission name `{{ $perm->name }}`

Putting it all together without any styling:

```
@foreach($permissionGroups as $module => $permissions)
    <h3>{{ Str::camel($module) }}</h3>
    <table>
        <thead>
        <tr>
            <th>Permisson</th>
            <th>Action</th>
        </tr>
        </thead>
        @foreach ($permissions as $perm)
            <tr>
                <td>{{ $perm->name }}</td>
                <td><input type="checkbox" name="permission[]" value="{{
$perm->id }}" @checked($role->hasPermissionTo($perm->name)) /></td>
            </tr>
        @endforeach
    </table>
@endforeach
```

Updating permissions to roles using syncPermissions

Once you've got an array of permissions you can update the permissions that are saved with a role. This will delete any permissions from the pivot table for the role that are not in the `$permissions` array.

```
$role->syncPermissions($permissions);
```

Custom namespaces

When you create a new module it also registers new custom namespace for `Lang`, `View` and `Config`. For example, if you create a new module named blog, it will also register new namespace/hint blog for that module. Then, you can use that namespace for calling `Lang`, `View` or `Config`. Following are some examples of its usage:

Calling Lang:

```
Lang::get('blog::group.name');

@trans('blog::group.name');
```

Calling View:

```
view('blog::index')

view('blog::partials.sidebar')
```

Calling Config:

```
Config::get('blog.name')
```

Your module will most likely contain what laravel calls resources, those contain configuration, views, translation files, etc.

In order for your module to correctly load and if wanted to publish them you need to let laravel know about them as in any regular package.

Note Those resources are loaded in the service provider generated with a module (using `module:make`), unless the `plain` flag is used, in which case you will need to handle this logic yourself.

Note Don't forget to change the paths, in the following code snippets a "Blog" module is assumed.

Configuration

```
$this->publishes([
    __DIR__.'/../config/config.php' => config_path('blog.php'),
], 'config');
$this->mergeConfigFrom(
    __DIR__.'/../config/config.php', 'blog'
);
```

Views

```
$viewPath = base_path('resources/views/modules/blog');

$sourcePath = __DIR__.'/../resources/views';

$this->publishes([
    $sourcePath => $viewPath
]);

$this->loadViewsFrom(array_merge(array_map(function ($path) {
    return $path . '/modules/blog';
}, \Config::get('view.paths')), [$sourcePath]), 'blog');
```

The main part here is the `loadViewsFrom` method call. If you don't want your views to be published to the laravel views folder, you can remove the call to the `$this->publishes()` call.

Language files

```
 $langPath = base_path('resources/lang/modules/blog');

if (is_dir($langPath)) {
    $this->loadTranslationsFrom($langPath, 'blog');
} else {
    $this->loadTranslationsFrom(__DIR__ .'/../lang', 'blog');
}
```

use in blade `{{ __(blog::foo) }}` will searched in:
`/lang/modules/en/foo.php`
`/Modules/Blog/lang/en/foo.php`

Factories

If you want to use laravel factories you will have to add the following in your service provider:

```
$this->app->singleton(Factory::class, function () {
    return Factory::construct(__DIR__ . '/database/factories');
});
```