LARAVEL MODULAR

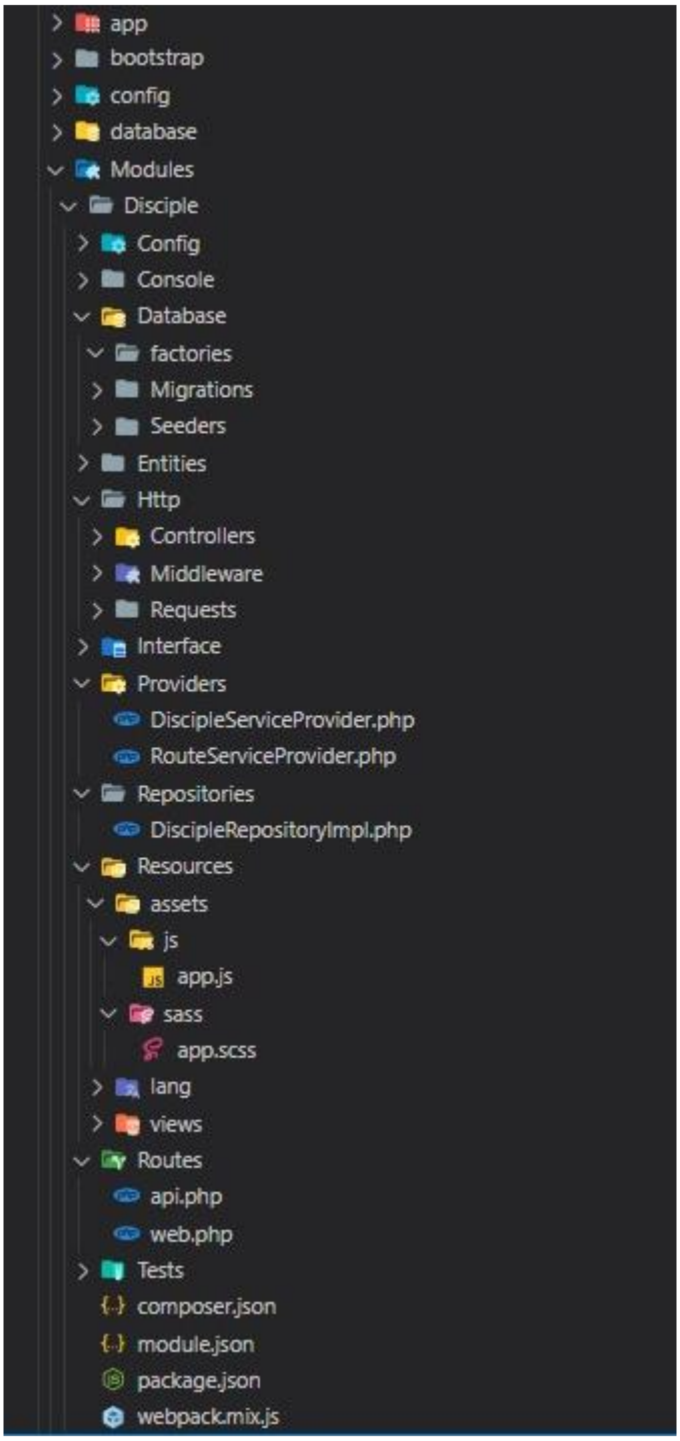
Poscar

Table of Contents

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

1. Laravel Modular

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



```
> app
> bootstrap
> config
> database
v Modules
  v Disciple
    > Config
    > Console
    v Database
      v factories
      > Migrations
      > Seeders
    > Entities
    v Http
      > Controllers
      > Middleware
      > Requests
    > Interface
    v Providers
        DiscipleServiceProvider.php
        RouteServiceProvider.php
    v Repositories
        DiscipleRepositoryImpl.php
    v Resources
      v assets
        v js
            app.js
        v sass
            app.scss
      > lang
      > views
    v Routes
        api.php
        web.php
    > Tests
      composer.json
      module.json
      package.json
      webpack.mix.js
```

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

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

- Controller
- Models
- Migrations
- Providers
- Repositories

-And other folders we might need.

2. Install Laravel Modules package

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

composer create-project laravel/Laravel^8.0 Student-management-system

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

Composer

To install through Composer, by run the following command:

composer require nwidart/laravel-modules

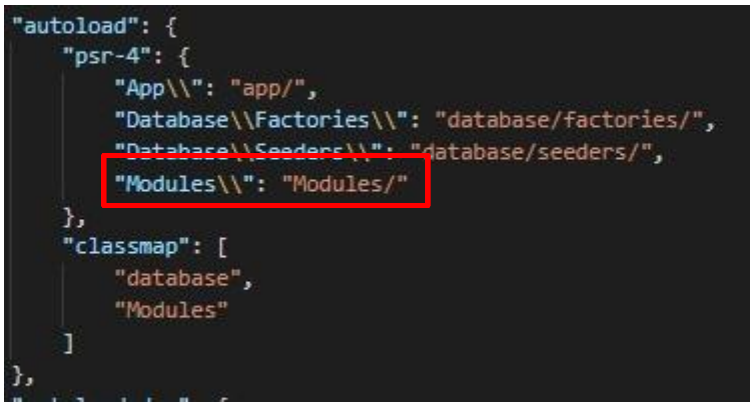
The package will automatically register a service provider.
Optionally, publish the package's configuration file by running:

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

Autoloading

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

"Modules\\": "Modules/"



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

Main app composer.json file

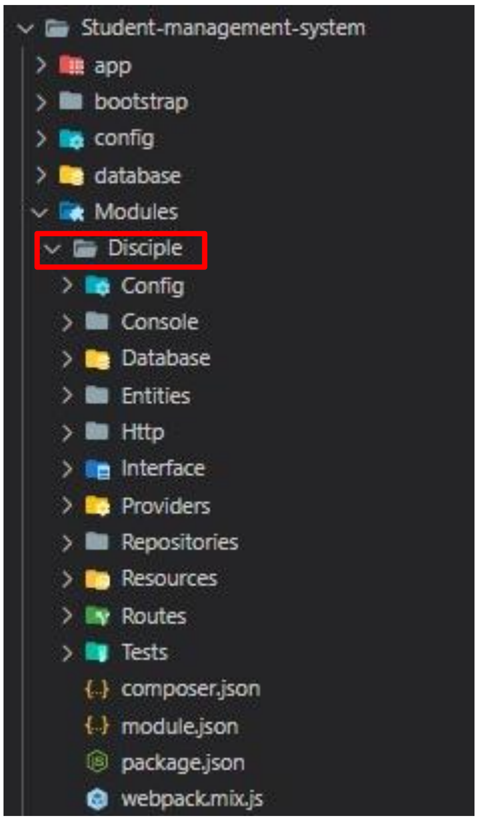
Don’t forget run this command:

composer dump-autoload

3. Create internal custom module

- Generate a new module:

php artisan module:make Disciple

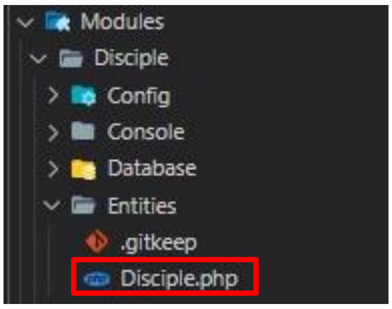


Project folder

Artisan commands

- Generate multiple modules at once:

php artisan module:make-model Disciple Disciple



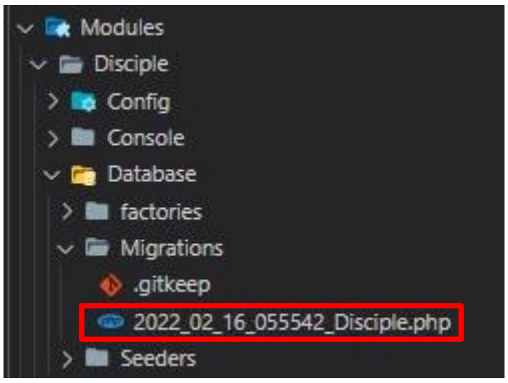
- Controller:

php artisan module:make-controller DiscipleController Disciple

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

- Migrate table:

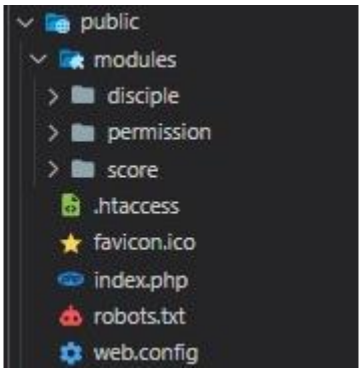
php artisan module:make-migration Disciple Disciple



Migrate the given module, or without a module an argument, migrate all modules.
php artisan module:migrate Disciple

- Publish Modules:

-**php artisan module:publish**
OR
- php artisan vendor:publish --tag=courier-config



After creating the modules, we want to use the modules work with the main project or another project. We use Repository Pattern in Laravel, using the Repository Pattern where each entity has a corresponding repository containing helper functions to interact with the database. While Module package doesn't provide this functionality out of the box, it is possible to use the Repository Pattern.

Create the Repository

Before we create a repository for the Disciple model, let's define an interface to specify all the methods which the repository must declare. Instead of relying directly on the repository class, our controller will depend on the interface.

In the **Modules/Disciple directory**, create a new folder called **Interface**. Then, in the Interface, create a new file called **DiscipleRepositoryInterface.php** and add the following code to it.

```php
<?php
namespace Disciple\Interface;
interface DiscipleRepositoryInterface
{
    public function getAllDisciples();
    public function createDisciple(array $discipleDetails);
    public function getDiscipleById($discipleId);
    public function updateDisciple($discipleId, array $newDetails);
    public function deleteDisciple($discipleId);
}
```

Next, in the **Modules** folder, create a new folder called **Repositories**. In this folder, create a new file called **DiscipleRepositoryImpl.php** and add the following code to it.

```php
<?php
namespace Disciple\Repositories;
use Modules\Disciple\Entities\Disciple;
use Disciple\Interface\DiscipleRepositoryInterface;

class DiscipleRepositoryImpl implements DiscipleRepositoryInterface
{
    public function getAllDisciples(){
        return Disciple::latest()->get();
    }
    public function createDisciple(array $discipleDetails){
        return Disciple::create($discipleDetails);
    }
    public function getDiscipleById($discipleId){
        return Disciple::findOrFail($discipleId);
    }
    public function updateDisciple($discipleId, array $newDetails){
        return Disciple::whereId($discipleId)->update($newDetails);
    }
    public function deleteDisciple($discipleId){
        Disciple::destroy($discipleId);
    }
}
```

Creating the controllers

With our repository in place, let's add some code to our controller.

Open **Modules/Disciple/Http/Controllers/DiscipleController.php** and update the code to match the following.

```php
class DiscipleController extends Controller
{
    private DiscipleRepositoryInterface $DiscipleRepository;
    public function __construct(DiscipleRepositoryInterface $discipleRepository){
        $this->discipleRepository = $discipleRepository;
    }
    public function index(): JsonResponse {
        return response()->json([
            'data' => $this->discipleRepository->getAllDisciples()
        ]);
    }
    public function store(Request $request): JsonResponse {
        $discipleDetails = $request->only(['user_id', 'dateWn', 'type', 'description']);
        return response()->json([
                'data' => $this->discipleRepository->createDisciple($discipleDetails)],Response::HTTP_CREATED
        );
    }
    public function show(Request $request): JsonResponse {
        $discipleId = $request->route('id');
        return response()->json([
            'data' => $this->discipleRepository->getDiscipleById($discipleId)
        ]);
    }
    public function update(Request $request): JsonResponse {
        $discipleId = $request->route('id');
        $discipleDetails = $request->only([ 'user_id', 'dateWn','type','description']);
        return response()->json([ 'message' => "Disciple updated",
            'data' => $this->discipleRepository->updateDisciple($discipleId, $discipleDetails)
        ]);
    }
    public function destroy(Request $request): JsonResponse {
        $discipleId = $request->route('id');
        $this->discipleRepository->deleteDisciple($discipleId);
        return response()->json(null, Response::HTTP_NO_CONTENT);
    }
}
```

The code injects an **DiscipleRepositoryInterface** instance via the constructor and uses the relevant object's methods in each controller method.

First, within the **index()** method, it calls the **getAllDisciples()** method defined in the **discipleRepository** to retrieve the list of Disciples and returns a response in JSON format.

Next, the **store()** method calls the **createDisciple()** method from the **discipleRepository** to create a new disciple. This takes the details of the Disciple that needs to be created as an array and returns a successful response afterward.

Within the **show()** method in the controller, it retrieves the unique Disciple Id from the route and passes it to the **getDiscipleById**() as a parameter. This fetches the details of the Disciple with a matching Id from the database and returns a response in JSON format.

Then, to update the details of an already created disciple, it calls the **updateDisciple()** method from the repository. This takes two parameters: the unique id of the disciple and the details that need to be updated.

Lastly, the **destroy()** method retrieves the unique id of a particular disciple from the route and calls the **deleteDisciple()** method from the repository to delete it.

Adding the routes

To map each method defined in the controller to specific routes, add the following code to *routes/api.php*.
Open **Modules/Disciple/Routes/api.php**

```
Route::get('/disciples', [DiscipleController::class, 'index']);
Route::get('/disciples/{id}', [DiscipleController::class, 'show']);
Route::post('/disciples', [DiscipleController::class, 'store']);
Route::put('/disciples/{id}', [DiscipleController::class, 'update']);
Route::delete('/disciples/{id}', [DiscipleController::class, 'destroy']);
```

Remember to include the import statement for the **DiscipleController:**

use Modules\Disciple\Http\Controllers\DiscipleController;

Bind the interface and the implementation

Now we need to do is bind **DiscipleRepositoryImpl** to **DiscipleRepositoryInterface** in Laravel service container;

We do this via Service Provider.

Open **Modules/Disciple/Providers/DiscipleServiceProvider.php** and update the register function to match the following.

```
class DiscipleServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            DiscipleRepositoryInterface::class,
            DiscipleRepositoryImpl::class
        );
    }
}
```

Remember to include the import statement or **DiscipleRepositoryImpl** and **DiscipleRepositoryInterface**:

use Disciple\Repositories\DiscipleRepositoryImpl;

use Disciple\Interface\DiscipleRepositoryInterface;

After, add the new Service Provider to the providers array in **Modules/Disciple/Config/config.php**

```
return [
    'name' => 'Disciple',
    'providers' => [
        Disciple\Repositories\DiscipleServiceProvider::class,
]
```

Finally, add the new Service Provider to the providers array in main project in **config** directory **app.php**

```
'providers' => [
       // …another declared provider
       Modules\Disciple\Providers\DiscipleServiceProvider::class,
       Modules\Disciple\Providers\RouteServiceProvider::class,
]
```

Test the application

Run the application using the following command.

$ php artisan serve

Run the following command to test the /api/disciples endpoint using URL:
http://localhost:8000/api/disciples

4. Attachments

1. https://nwidart.com/laravel-modules/v6/introduction

Thank you!