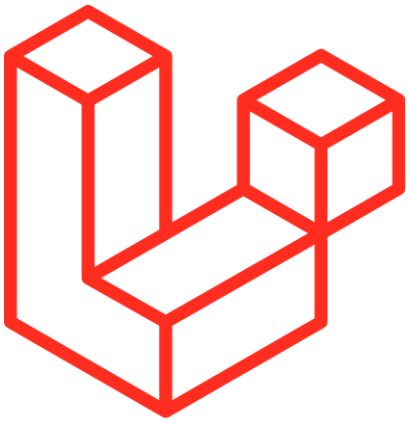
LARAVEL PACKAGE



Poscar

Table of contents

What are packages?

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

1. Creating Laravel Package

Prerequisites

Before we can start creating our package, it is expected that you already have Composer and Git installed on your system. It is also required that you can create a remote repository to any VCS service like GitLab or GitHub. For this, we are going to use GitLab and Laravel 8.6.0

Package composer.json

Inside your command prompt navigate to the folder with your package name. In our case: htdocs/LARAVEL/Laravel Package/Permission/ and running the following command to create the empty composer project:

```
$ composer init
```

```
C:\xampp\Xampp\htdocs\LARAVEL\Laravel Package\Permission>composer init

Welcome to the Composer config generator

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

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

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Add PSR-4 autoload mapping? Maps namespace "Module\Permission" to the entered relative path. [src/, n to skip]:
{
    "name": "module/permission",
    "description": "Permissin package",
    "type": "package",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "Module\\Permission\\": "src/"
        }
    },
    "minimum-stability": "dev",
    "require": {}
}
```

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

C:\xampp\Xampp\htdocs\LARAVEL\Laravel Package\Permission>
```

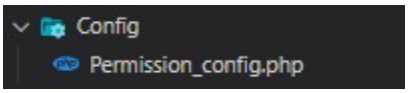
So, our folder structure will be in **LARAVEL/Laravel Package/Permission**

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

```
{
    "name": "module/permission",
    "description": "Permission library",
    "keywords": ["Laravel-package","Permission", "Library"],
    "minimum-stability": "dev",
    "type": "library",
    "license": "MIT",
    "authors": [
        {
            "name": "Lyden Chai",
            "email": "lyden.chai@gmail.com"
        }
    ],
    "autoload": {
        "psr-4": {
            "Module\\Permission\\": ""
        }
    },
    "extra": {
        "laravel": {
            "providers": [
                "Module\\Permission\\Providers\\PermissionServiceProvider"
            ]
        }
    },
    "require": {
        "php": "^7.3|^8.0",
        "illuminate/config": "^8.0 || ^9.0",
        "illuminate/database": "^8.53 || ^9.0",
        "illuminate/support": "^8.0 || ^9.0",
        "laravel/framework": "^8.54",
        "laravel/sanctum": "^2.11",
        "laravel/tinker": "^2.5"
    }
}
```

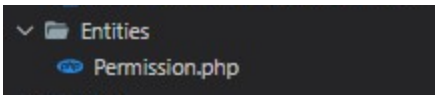
Now let's create folder that we need:

Config directory:
The config directory, as the name implies, contains all of application's configuration files.
In this directory we have **Permission_config.php**

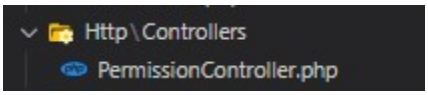


Database directory:
The database directory contains database migrations, model factories
In Database/Migrateions directory we have **create_permissions_table_.php.stub**

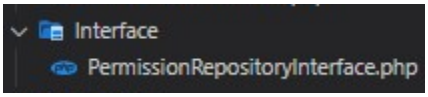
Entities directory:
The Entities directory contains all of Eloquent model classes.
In this directory we have **Permission.php**



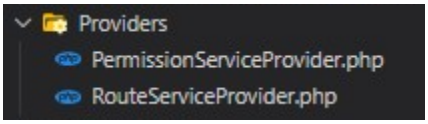
Http directory:
The Http directory contains controllers, middleware, and form requests.
In this directory we have **PermissionController.php**



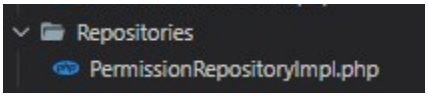
Interface directory:
In Interface directory we have **PermissionRepositoryInterface.php**



Providers directory:
The Providers directory contains all of the service providers for our package.
In this directory we have **PermissionServiceProvider.php** and **RouteServiceProvider.php**

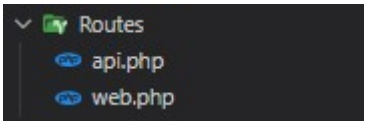


Repositorites directory:
In Repositories directory we have **PermissionRepositoryImpl.php**

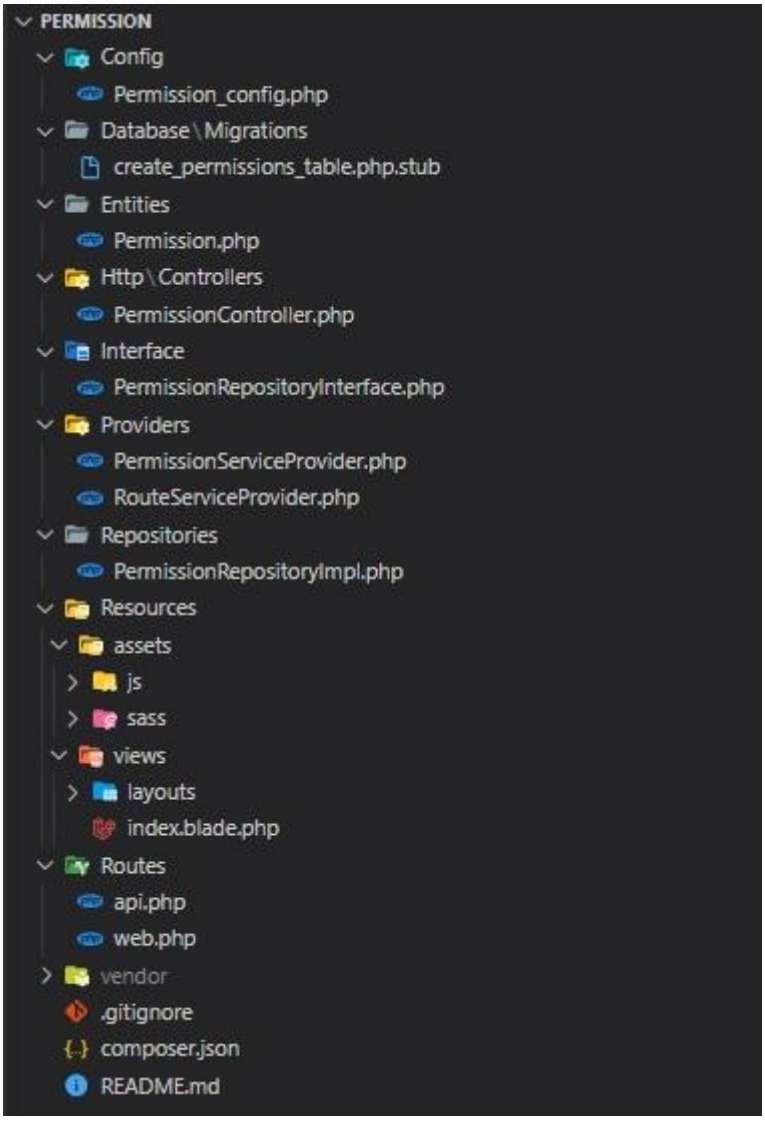


Resource directory:
In Resource directory we have Assets and View folder

Route directory:
The routes directory contains all of the route definitions for our package.
In this directory we have **api.php** and **web.php**



So, here is our folder structure we created in **LARAVEL/Laravel Package/Permission**



After creating the project structure, we want to use this package work with the other project. We use Repository Pattern in Laravel.

Create the repository

Before we create a repository for the Permission package 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 **Permission/Interface/PermissionRepositoryInterface.php** and add the following code to it.

```php
<?php

namespace Module\Permission\Interface;

interface PermissionRepositoryInterface
{
    public function all();
    public function save(array $details);
    public function find($id);
    public function update($id, array $newDetails);
    public function delete($id);
}
```

Next, in **Permissioin/Reppositories/PermissionRepositoryImpl.php** add the following code to it.

```php
<?php
namespace Module\Permission\Repositories;
use Module\Permission\Entities\Permission;
use Module\Permission\Interface\PermissionRepositoryInterface;
class PermissionRepositoryImpl implements PermissionRepositoryInterface
{
    public function all()
    {
        return Permission::latest()->get();
    }
    public function save(array $details)
    {
        return Permission::create($details);
    }
    public function find($id)
    {
        return Permission::findOrFail($id);
    }
    public function update($id, array $newDetails)
    {
        return Permission::whereId($id)->update($newDetails);
    }
    public function delete($id)
    {
        Permission::destroy($id);
    }
}
```

Create the controller

With our repository in place, let's add some code to our controller.
Open **Permission/Http/Controllsers/PermissionController.php** and update the code to match the following.

```php
<?php
namespace Modules\Permission\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Http\Response;
use Illuminate\Http\JsonResponse;
use Illuminate\Routing\Controller;
use Modules\Permission\Entitites\Permission;
use Illuminate\Contracts\Support\Renderable;
use Permission\Interface\PermissionRepositoryInterface;
class PermissionController extends Controller
{
    private PermissionRepositoryInterface $PermissionRepository;

    public function __construct(PermissionRepositoryInterface $permissionRepository)
    {
        $this->permissionRepository = $permissionRepository;
    }
    public function index(): JsonResponse
    {
        return response()->json([
            'data' => $this->permissionRepository->all()
        ]);
    }
    public function store(Request $request): JsonResponse
    {
        $details = $request->only([
            'user_id',
            'start_At',
            'end_At',
            'type',
            'description',
        ]);
        return response()->json(
            [
                'message' => 'Permission created',
                'data' => $this->permissionRepository->save($details)
            ],
            Response::HTTP_CREATED
        );
    }
    public function show(Request $request): JsonResponse
    {
        $id = $request->route('id');
        return response()->json([
            'data' => $this->permissionRepository->find($id)
```

```
        ]);
    }
    public function update(Request $request): JsonResponse
    {
        $id = $request->route('id');
        $details = $request->only([
            'user_id',
            'start_At',
            'end_At',
            'type',
            'description',
        ]);
        return response()->json([
            'message' => "Permission updated",
            'data' => $this->permissionRepository->update($id, $details)
        ]);
    }
    public function destroy(Request $request): JsonResponse
    {
        $id = $request->route('id');
        $this->permissionRepository->delete($id);
        return response()->json([
            'message' => 'Permission deleted'
        ], Response::HTTP_NO_CONTENT);
    }
}
```

Adding the routes

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

```
// Permission Public Route
Route::get('/permissions', [PermissionController::class, 'index']);
Route::get('/permissions/{id}', [PermissionController::class, 'show']);

// Permission Private Route
Route::post('/permissions', [PermissionController::class, 'store']);
Route::put('/permissions/{id}', [PermissionController::class, 'update']);
Route::delete('/permissions/{id}', [PermissionController::class, 'destroy']);
```

Remember to include the import statement for the PermissionController:

```
use Module\Permission\Http\Controllers\PermissionController;
```

Binding the interface and the implement

Now we need to do is bind **PermissionRepositoryImpl** to **PermissionRepositoryInterface** in Laravel service container; We do this via Service Provider.

Open **Permission/Providers/PermissioinServiceProvider.php** and update the register function to match the following.

And in **Permission/Providers/PermissioinServiceProvider.php** we also have **registerPublishables()** method for publish configuration and migration.

```
<?php
namespace Module\Permission\Providers;
use Illuminate\Support\ServiceProvider;
use Module\Permission\Repositories\PermissionRepositoryImpl;
use Module\Permission\Interface\PermissionRepositoryInterface;
class PermissionServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            PermissionRepositoryInterface::class,
            PermissionRepositoryImpl::class
        );
    }
    public function boot()
    {
        $this->registerPublishables();
    }
    protected function registerPublishables(): void
    {
        if ($this->app->runningInConsole()) {
            $this->publishes([
                __DIR__ . '/../Config/Permission_config.php' =>
 config_path('Permission_config.php'),
            ], 'config');
        }

        if (!class_exists('CreatePermissionsTable')) {
            $this->publishes([
                __DIR__ . '/../database/migrations/create_permissions_table.php.stub' =>
 database_path('migrations/' . date('Y_m_d_His', time()) . '_create_permissions_table.php'),
            ], 'migrations');
        }
    }
}
```

In **Permission/Providers/RouteServiceProvider.php** we have **mapApiRoutes()** method to define the “api” route for the package.

```
<?php
namespace Module\Permission\Providers;
use Illuminate\Support\Facades\Route;
use Illuminate\Foundation\Support\Providers\RouteServiceProvider as ServiceProvider;
```

```
class RouteServiceProvider extends ServiceProvider
{
    protected $namespace = 'Module\Permission\Http\Controllers';
    public function map()
    {
        $this->mapApiRoutes();
    }
    protected function mapApiRoutes()
    {
        Route::prefix('api')
            ->middleware('api')
            ->namespace($this->namespace)
            ->group(__DIR__ . '/../Routes/api.php');
    }
}
```

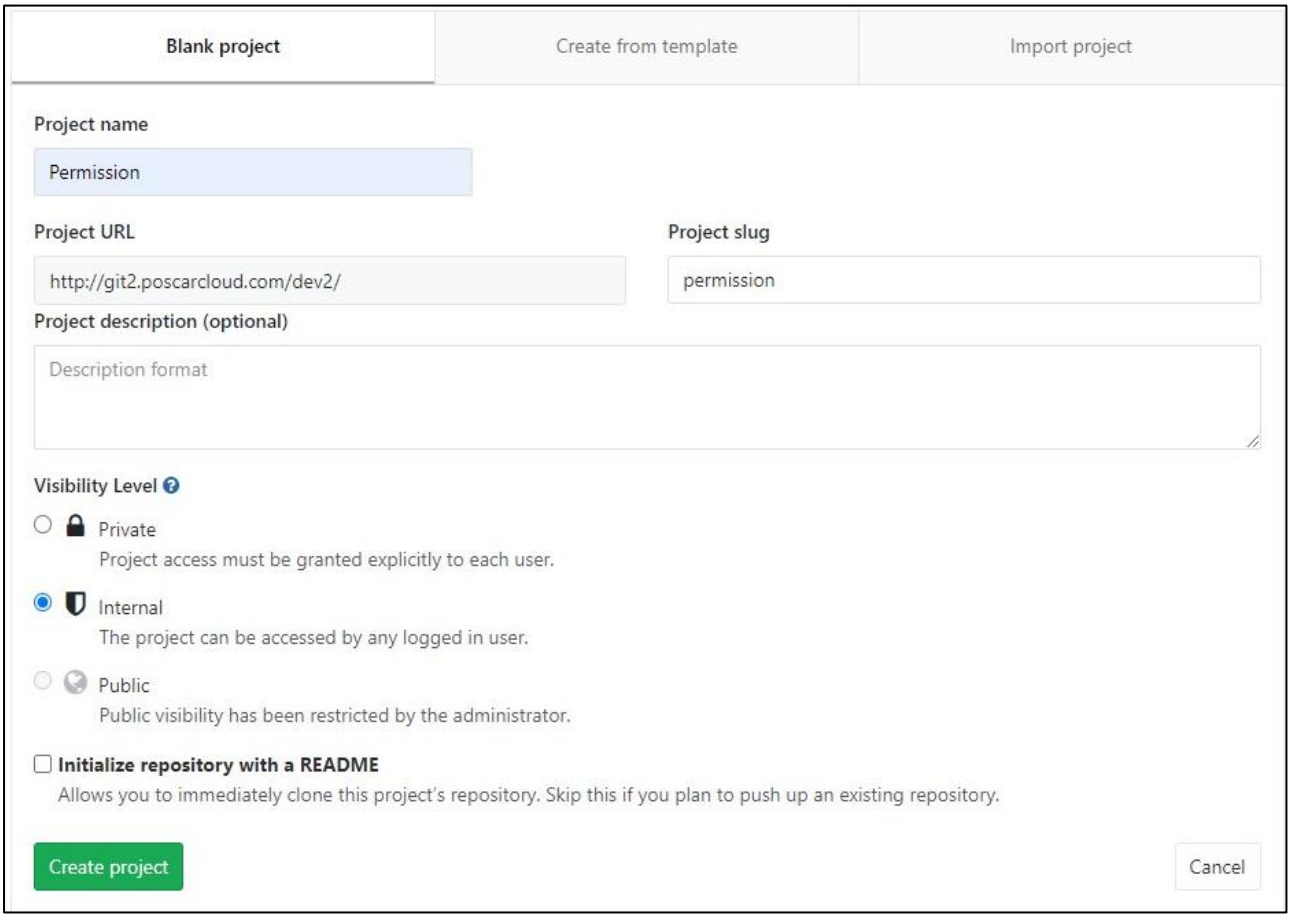
After, add the new Service Provider to the providers array in **Permission/Config/config.php**

```
<?php
return [
    'name' => 'Permission',
    'providers' => [
       Module\Permission\Providers\PermissionServiceProvider::class,
        Module\Permission\Providers\RouteServiceProvider::class,
    ]
];
```

2. Publishing Package to GitLab

Create GitLab repository

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



Initialize git, commit changes, and tag changeset with semantic versioning by running the following commands.

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

Push local source code to remote GitLab serve

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

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

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

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

3. Importing Laravel Package

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

Require our package in compoer.json

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

In project's **composer.json** file we have to add:

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

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

```
$ composer update
```

Or we can install through Composer, by run the following command:

```
$ composer require module/permission
```

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

```
$ composer update
```

We now should find our package in the vendor directory of our project. We should be able to use our package like any other packages that we require in our project.

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

```
$ php artisan vendor:publish –provider="module\permission\Provider\PermissionServiceProvider"
```

If you want to use route's package, add the following to the providers array in config/app.php. This provider must be registered as the last service provider on the providers array:

```php
'providers' => [
   // …another declared provider
   Module\Permission\Providers\RouteServiceProvider::class,
]
```

Choosing a Package Version

When other developers are using your package, there will come a time that they will start raising bugs, improvements or suggest new features. You as the maintainer of the package will have to fix that bug and release a new version. Now, all developers using your package will be informed about your fix in your repository documentations.

The thing is, whenever a new version is released, developers will have the option to upgrade to that release if they are affected by the bug or choose not to. This can be done in Composer very easily by just updating your project's composer.json file with the desired version, run composer update, run your tests, and redeploy. If the new release doesn't work, you may rollback to previous versions easily. This is one of those perks of having Composer manage your package dependencies and it's really useful.

Summary

Packages make our daily work as developers much easier, especially if there are a lot of ready-made packages that we can use in our projects. Sharing packages makes our creation live longer and gets a chance to be appreciated by others for the efforts we've put into it. Let's start building your packages now!

4. Attachments

1. https://composer.json.jolicode.com
2. https://www.twilio.com/blog/repository-pattern-in-laravel-application
3. https://github.com/lydenchai/Laravel-Package/
4. https://github.com/spatie/laravel-medialibrary
5. https://laravelpackage.com/02-development-environment.html#orchestra-testbench

Thank you!