Package Development

Introduction

Packages are the primary way of adding functionality to Laravel. Packages might be anything from a great way to work with dates like Carbon, or an entire BDD testing framework like Behat.

There are different types of packages. Some packages are stand-alone, meaning they work with any PHP framework. Carbon and Behat are examples of stand-alone packages. Any of these packages may be used with Laravel by requesting them in your `composer.json` file.

On the other hand, other packages are specifically intended for use with Laravel. These packages may have routes, controllers, views, and configuration specifically intended to enhance a Laravel application. This guide primarily covers the development of those packages that are Laravel specific.

A Note On Facades

When writing a Laravel application, it generally does not matter if you use contracts or facades since both provide essentially equal levels of testability. However, when writing packages, your package will not typically have access to all of Laravel's testing helpers. If you would like to be able to write your package tests as if they existed inside a typical Laravel application, you may use the Orchestral Testbench package.

Package Discovery

In a Laravel application's `config/app.php` configuration file, the `providers` option defines a list of service providers that should be loaded by Laravel. When someone installs your package, you will typically want your service provider to be included in this list. Instead of requiring users to manually add your service provider to the list, you may define the provider in the `extra` section of your package's `composer.json` file. In addition to service providers, you may also list any facades you would like to be registered:

```
"extra": {
    "laravel": {
        "providers": [
            "Barryvdh\\Debugbar\\ServiceProvider"
        ],
        "aliases": {
            "Debugbar": "Barryvdh\\Debugbar\\Facade"
        }
    }
},
```

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

Opting Out Of Package Discovery

If you are the consumer of a package and would like to disable package discovery for a package, you may list the package name in the `extra` section of your application's `composer.json` file:

```
"extra": {
```

```
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

You may disable package discovery for all packages using the `*` character inside of your application's `dont-discover` directive:

```
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
},
```

Service Providers

Service providers are the connection points between your package and Laravel. A service provider is responsible for binding things into Laravel's service container and informing Laravel where to load package resources such as views, configuration, and localization files.

A service provider extends the `Illuminate\Support\ServiceProvider` class and contains two methods: `register` and `boot`. The base `ServiceProvider` class is located in the `illuminate/support` Composer package, which you should add to your own package's dependencies. To learn more about the structure and purpose of service providers, check out their documentation.

Resources

Configuration

Typically, you will need to publish your package's configuration file to the application's own `config` directory. This will allow users of your package to easily override your default configuration options. To allow your configuration files to be published, call the `publishes` method from the `boot` method of your service provider:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__.'/path/to/config/courier.php' => config_path('courier.php'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` command, your file will be copied to the specified publish location. Once your configuration has been published, its values may be accessed like any other configuration file:

```
$value = config('courier.option');
```

You should not define Closures in your configuration files. They can not be serialized correctly when users execute the `config:cache` Artisan command.

Default Package Configuration

You may also merge your own package configuration file with the application's published copy. This will allow your users to define only the options they actually want to override in the published copy of the configuration. To merge the configurations, use the `mergeConfigFrom` method within your service provider's `register` method:

```
/**
 * Register bindings in the container.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
```

```
        __DIR__.'/path/to/config/courier.php', 'courier'
    );
}
```

This method only merges the first level of the configuration array. If your users partially define a multi-dimensional configuration array, the missing options will not be merged.

Routes

If your package contains routes, you may load them using the `loadRoutesFrom` method. This method will automatically determine if the application's routes are cached and will not load your routes file if the routes have already been cached:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__.'/routes.php');
}
```

Migrations

If your package contains database migrations, you may use the `loadMigrationsFrom` method to inform Laravel how to load them. The `loadMigrationsFrom` method accepts the path to your package's migrations as its only argument:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/path/to/migrations');
}
```

Once your package's migrations have been registered, they will automatically be run when the `php artisan migrate` command is executed. You do not need to export them to the application's main `database/migrations` directory.

Translations

If your package contains translation files, you may use the `loadTranslationsFrom` method to inform Laravel how to load them. For example, if your package is named `courier`, you should add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');
}
```

Package translations are referenced using the `package::file.line` syntax convention. So, you may load the `courier` package's `welcome` line from the `messages` file like so:

```
echo trans('courier::messages.welcome');
```

Publishing Translations

If you would like to publish your package's translations to the application's `resources/lang/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package paths and their desired publish locations. For example, to publish the translation files for the `courier` package, you may do the following:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
```

```
public function boot()
{
    $this->loadTranslationsFrom(__DIR__.'/path/to/translations', 'courier');

    $this->publishes([
        __DIR__.'/path/to/translations' => resource_path('lang/vendor/courier'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's translations will be published to the specified publish location.

Views

To register your package's views with Laravel, you need to tell Laravel where the views are located. You may do this using the service provider's `loadViewsFrom` method. The `loadViewsFrom` method accepts two arguments: the path to your view templates and your package's name. For example, if your package's name is `courier`, you would add the following to your service provider's `boot` method:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');
}
```

Package views are referenced using the `package::view` syntax convention. So, once your view path is registered in a service provider, you may load the `admin` view from the `courier` package like so:

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

Overriding Package Views

When you use the `loadViewsFrom` method, Laravel actually registers two locations for your views: the application's `resources/views/vendor` directory and the directory you specify. So, using the `courier` example, Laravel will first check if a custom version of the view has been provided by the developer in `resources/views/vendor/courier`. Then, if the view has not been customized, Laravel will search the package view directory you specified in your call to `loadViewsFrom`. This makes it easy for package users to customize / override your package's views.

Publishing Views

If you would like to make your views available for publishing to the application's `resources/views/vendor` directory, you may use the service provider's `publishes` method. The `publishes` method accepts an array of package view paths and their desired publish locations:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');

    $this->publishes([
        __DIR__.'/path/to/views' => resource_path('views/vendor/courier'),
    ]);
}
```

Now, when users of your package execute Laravel's `vendor:publish` Artisan command, your package's views will be copied to the specified publish location.

Commands

To register your package's Artisan commands with Laravel, you may use the `commands` method. This method expects an array of command class names. Once the commands have been registered, you may execute them using the Artisan CLI:

```
/**
 * Bootstrap the application services.
```

```
     *
     * @return void
     */
    public function boot()
    {
        if ($this->app->runningInConsole()) {
            $this->commands([
                FooCommand::class,
                BarCommand::class,
            ]);
        }
    }
```

Public Assets

Your package may have assets such as JavaScript, CSS, and images. To publish these assets to the application's `public` directory, use the service provider's `publishes` method. In this example, we will also add a `public` asset group tag, which may be used to publish groups of related assets:

```
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        $this->publishes([
            __DIR__.'/path/to/assets' => public_path('vendor/courier'),
        ], 'public');
    }
```

Now, when your package's users execute the `vendor:publish` command, your assets will be copied to the specified publish location. Since you will typically need to overwrite the assets every time the package is updated, you may use the `--force` flag:

```
php artisan vendor:publish --tag=public --force
```

Publishing File Groups

You may want to publish groups of package assets and resources separately. For instance, you might want to allow your users to publish your package's configuration files without being forced to publish your package's assets. You may do this by "tagging" them when calling the `publishes` method from a package's service provider. For example, let's use tags to define two publish groups in the `boot` method of a package service provider:

```
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        $this->publishes([
            __DIR__.'/../config/package.php' => config_path('package.php')
        ], 'config');

        $this->publishes([
            __DIR__.'/../database/migrations/' => database_path('migrations')
        ], 'migrations');
    }
```

Now your users may publish these groups separately by referencing their tag when executing the `vendor:publish` command:

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

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.

Scout

Socialite