1-The entry point for all requests to a Laravel application is the public/index.php file. All requests are directed to this file by your web server (Apache / Nginx). The index.php file is a starting point for loading the rest of the framework.

It auto loads the auto loader files via bootstrap/autoload.phpwhich is generated by composer.

Then it retrieves an instance of the Laravel application frombootstrap/app.php file. Laravel itself creates an instance of the application.

Next step will occur on the Kernel part of the application.

The incoming request is sent to either the HTTP kernel or the console kernel, depending on the type of request that is entering into the application .These two kernels serve as the central location that all requests flow through.

HTTP kernel, which is placed in app/Http/Kernel.php. It just receive a Request and return a Response. Bootstrappers that are defined by the Kernel class, which configures error handling, configure logging, detect environments and other tasks to be done before the request handled.

HTTP Kernel will define the list of middleware that are passed through before handled by application.

Next step of the kernel is to load service providers as part of the bootstrapping action. Providers that are needed for the application are placed in config/app.php configuration file.

While the register method calls, all the providers will be registered. Once all providers are registered, then boot method will be called.

Once the application have been bootstrapped and all service providers are registered and booted, the request will be handed over to the router for dispatching. The router will dispatch the request to a route or controller, as well as run any route specific middleware.

Now request will be dispatched by the Router and it will end up with the views as shown below:

Router will direct the HTTP Request to a Controller or return a view or responses directly by omitting the controller. These routes will be placed in routes/

Controller app/controllers/ performs specific actions and sends data to a View.

View resource/views/ formats the data appropriately, providing the HTTP Response.

After sending response kernel terminate the request. And that’s all about request life cycle of Laravel.

Service Container

It is basically the feature that does the dependency resolution / dependency injections. It contains all the bindings that we want to inject as dependency.

Service Provider

Basically a module registry system. Let’s say we want to organize our dependency bindings, middlewares, event listeners etc, then we create a service provider and register related things in it. So even though the Service Container is responsible for injecting dependencies, we actually bind the dependencies in Service Provider.

Facade

Some build in static classes that we can use easily to do common tasks such as access session, cache etc.

Please feel free to leave a comment if you have any feedback, questions or want me to write about another PHP/Laravel related topic.

2-Props to SitePoint for sharing such informative and helpful knowledge about facades in Laravel.

The facade pattern is a software design pattern that is often used in object-oriented programming.

A facade is a class wrapping a complex library to provide a simpler and more readable interface to it.

Facades in Laravel

Facades provide a "static" interface to classes that are available in the application's service container. Laravel ships with many facades which provide access to almost all of Laravel's features. Laravel facades serve as "static proxies" to underlying classes in the service container, providing the benefit of a terse, expressive syntax while maintaining more testability and flexibility than traditional static methods.

How Facades Are implemented in Laravel

Every service inside the container has a unique name. In a Laravel application, to access a service directly from the container, we can use the App::make() method or the app() helper function.

<?php

App::make('some\_service')->methodName();

In Laravel, all services have a facade class. These facade classes extend the base Facade class which is part of the Illuminate/Support package. The only thing that they need to implement is the getFacadeAccessor method, which returns the service name inside the container.

2-When To Use Facades

Facades have many benefits. They provide a terse, memorable syntax that allows you to use Laravel's features without remembering long class names that must be injected or configured manually. Furthermore, because of their unique usage of PHP's dynamic methods, they are easy to test.

However, some care must be taken when using facades. The primary danger of facades is class "scope creep". Since facades are so easy to use and do not require injection, it can be easy to let your classes continue to grow and use many facades in a single class. Using dependency injection, this potential is mitigated by the visual feedback a large constructor gives you that your class is growing too large. So, when using facades, pay special attention to the size of your class so that its scope of responsibility stays narrow. If your class is getting too large, consider splitting it into multiple smaller classes.

Facades Vs. Dependency Injection

One of the primary benefits of dependency injection is the ability to swap implementations of the injected class. This is useful during testing since you can inject a mock or stub and assert that various methods were called on the stub.

Typically, it would not be possible to mock or stub a truly static class method. However, since facades use dynamic methods to proxy method calls to objects resolved from the service container, we actually can test facades just as we would test an injected class instance. For example, given the following route:

3-IntroductionService providers are the central place of all Laravel application bootstrapping. Your own application, as well as all of Laravel's core services, are bootstrapped via service providers.

But, what do we mean by "bootstrapped"? In general, we mean registering things, including registering service container bindings, event listeners, middleware, and even routes. Service providers are the central place to configure your application.

If you open the config/app.php file included with Laravel, you will see a providers array. These are all of the service provider classes that will be loaded for your application. By default, a set of Laravel core service providers are listed in this array. These providers bootstrap the core Laravel components, such as the mailer, queue, cache, and others. Many of these providers are "deferred" providers, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

In this overview, you will learn how to write your own service providers and register them with your Laravel application.

If you would like to learn more about how Laravel handles requests and works internally, check out our documentation on the Laravel request lifecycle.

Writing Service ProvidersAll service providers extend the Illuminate\Support\ServiceProvider class. Most service providers contain a register and a boot method. Within the register method, you should only bind things into the service container. You should never attempt to register any event listeners, routes, or any other piece of functionality within the register method.

The Artisan CLI can generate a new provider via the make:provider command:

php artisan make:provider RiakServiceProvider

4-The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods,When To Use The Container

Thanks to zero configuration resolution, you will often type-hint dependencies on routes, controllers, event listeners, and elsewhere without ever manually interacting with the container. For example, you might type-hint the Illuminate\Http\Request object on your route definition so that you can easily access the current request. Even though we never have to interact with the container to write this code, it is managing the injection of these dependencies behind the scenes,In many cases, thanks to automatic dependency injection and facades, you can build Laravel applications without ever manually binding or resolving anything from the container. So, when would you ever manually interact with the container? Let's examine two situations.

First, if you write a class that implements an interface and you wish to type-hint that interface on a route or class constructor, you must tell the container how to resolve that interface. Secondly, if you are writing a Laravel package that you plan to share with other Laravel developers, you may need to bind your package's services into the container.

5-Introduction

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the validate method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user. By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:s :

6-Laravel's Illuminate\Http\Request class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.

Interacting With The Request

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the Illuminate\Http\Request class on your route closure or controller method. The incoming request instance will automatically be injected by the Laravel service container:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;

use Illuminate\Http\Request;

class UserController extends Controller

{

    /\*\*

     \* Store a new user.

     \*/

    public function store(Request $request): RedirectResponse

    {

        $name = $request->input('name');

        // Store the user...

        return redirect('/users');

    }

}

As mentioned, you may also type-hint the Illuminate\Http\Request class on a route closure. The service container will automatically inject the incoming request into the closure when it is executed:

use Illuminate\Http\Request;

Route::get('/', function (Request $request) {

    // ...

});

References

<https://mohasin-dev.medium.com/laravel-architectural-concepts-d8bc44162a3f#:~:text=Controller%20app%2Fcontrollers%2F%20performs%20specific%20actions%20and%20sends%20data,that%E2%80%99s%20all%20about%20request%20life%20cycle%20of%20Laravel>

<https://laravel.com/docs/10.x/lifecycle>

<https://stackoverflow.com/questions/42223986/what-is-facades-used-in-laravel>

<https://laravel.com/docs/10.x/facades>

<https://laravel.com/docs/10.x/providers>

<https://laravel.com/docs/10.x/container>

<https://laravel.com/docs/10.x/validation>

<https://laravel.com/docs/10.x/requests>