Request Lifecycle

Introduction

When using any tool in the "real world", you feel more confident if you understand how that tool works. Application development is no different. When you understand how your development tools function, you feel more comfortable and confident using them.

The goal of this document is to give you a good, high-level overview of how the Laravel framework works. By getting to know the overall framework better, everything feels less "magical" and you will be more confident building your applications. If you don't understand all of the terms right away, don't lose heart! Just try to get a basic grasp of what is going on, and your knowledge will grow as you explore other sections of the documentation.

Lifecycle Overview

First Steps

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) configuration. The `index.php` file doesn't contain much code. Rather, it is a starting point for loading the rest of the framework.

The `index.php` file loads the Composer generated autoloader definition, and then retrieves an instance of the Laravel application from `bootstrap/app.php`. The first action taken by Laravel itself is to create an instance of the application / service container.

HTTP / Console Kernels

Next, the incoming request is sent to either the HTTP kernel or the console kernel, using the `handleRequest` or `handleCommand` methods of the application instance, depending on the type of request entering the application. These two kernels serve as the central location through which all requests flow. For now, let's just focus on the HTTP kernel, which is an instance of `Illuminate\Foundation\Http\Kernel`.

The HTTP kernel defines an array of `bootstrappers` that will be run before the request is executed. These bootstrappers configure error handling, configure logging, detect the application environment, and perform other tasks that need to be done before the request is actually handled. Typically, these classes handle internal Laravel configuration that you do not need to worry about.

The HTTP kernel is also responsible for passing the request through the application's middleware stack. These middleware handle reading and writing the HTTP session, determining if the application is in maintenance mode, verifying the CSRF token, and more. We'll talk more about these soon.

The method signature for the HTTP kernel's `handle` method is quite simple: it receives a `Request` and returns a `Response`. Think of the kernel as being a big black box that represents your entire application. Feed it HTTP requests and it will return HTTP responses.

Service Providers

One of the most important kernel bootstrapping actions is loading the service providers for your application. Service providers are responsible for bootstrapping all of the framework's various components, such as the database, queue, validation, and routing components.

Laravel will iterate through this list of providers and instantiate each of them. After instantiating the providers, the `register` method will be called on all of the providers. Then, once all of the providers have been registered, the `boot` method will be called on each provider. This is so service providers may depend on every container binding being registered and available by the time their `boot` method is executed.

Essentially every major feature offered by Laravel is bootstrapped and configured by a service provider. Since they bootstrap and configure so many features offered by the framework, service providers are the most important aspect of the entire Laravel bootstrap process.

While the framework internally uses dozens of service providers, you also have the option to create your own. You can find a list of the user-defined or third-party service providers that your application is using in the `bootstrap/providers.php` file.

Routing

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

Middleware provide a convenient mechanism for filtering or examining HTTP requests entering your application. For example, Laravel includes a middleware that verifies if the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application. Some middleware are assigned to all routes within the application, like `PreventRequestsDuringMaintenance`, while some are only assigned to specific routes or route groups. You can learn more about middleware by reading the complete middleware documentation.

If the request passes through all of the matched route's assigned middleware, the route or controller method will be executed and the response returned by the route or controller method will be sent back through the route's chain of middleware.

Finishing Up

Once the route or controller method returns a response, the response will travel back outward through the route's middleware, giving the application a chance to modify or examine the outgoing response.

Finally, once the response travels back through the middleware, the HTTP kernel's `handle` method returns the response object to the `handleRequest` of the application instance, and this method calls the `send` method on the returned response. The `send` method sends the response content to the user's web browser. We've now completed our journey through the entire Laravel request lifecycle!

Focus on Service Providers

Service providers are truly the key to bootstrapping a Laravel application. The application instance is created, the service providers are registered, and the request is handed to the bootstrapped application. It's really that simple!

Having a firm grasp of how a Laravel application is built and bootstrapped via service providers is very valuable. Your application's user-defined service providers are stored in the `app/Providers` directory.

By default, the `AppServiceProvider` is fairly empty. This provider is a great place to add your application's own bootstrapping and service container bindings. For large applications, you may wish to create several service providers, each with more granular bootstrapping for specific services used by your application.