Error Handling

Introduction

When you start a new Laravel project, error and exception handling is already configured for you. The `App\Exceptions\Handler` class is where all exceptions triggered by your application are logged and then rendered back to the user. We'll dive deeper into this class throughout this documentation.

Configuration

The `debug` option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your `.env` file.

For local development, you should set the `APP_DEBUG` environment variable to `true`. In your production environment, this value should always be `false`. If the value is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.

The Exception Handler

The Report Method

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains two methods: `report` and `render`. We'll examine each of these methods in detail. The `report` method is used to log exceptions or send them to an external service like Flare, Bugsnag or Sentry. By default, the `report` method passes the exception to the base class where the exception is logged. However, you are free to log exceptions however you wish.

For example, if you need to report different types of exceptions in different ways, you may use the PHP `instanceof` comparison operator:

```
/**
 * Report or log an exception.
 *
 * This is a great spot to send exceptions to Flare, Sentry, Bugsnag, etc.
 *
 * @param  \Exception  $exception
 * @return void
 */
public function report(Exception $exception)
{
    if ($exception instanceof CustomException) {
        //
    }

    parent::report($exception);
}
```

Instead of making a lot of `instanceof` checks in your `report` method, consider using reportable exceptions

Global Log Context

If available, Laravel automatically adds the current user's ID to every exception's log message as contextual data. You may define your own global contextual data by overriding the `context` method of your application's `App\Exceptions\Handler` class. This information will be included in every exception's log message written by your application:

```php
/**
 * Get the default context variables for logging.
 *
 * @return array
 */
protected function context()
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

The `report` Helper

Sometimes you may need to report an exception but continue handling the current request. The `report` helper function allows you to quickly report an exception using your exception handler's `report` method without rendering an error page:

```php
public function isValid($value)
{
    try {
        // Validate the value...
    } catch (Exception $e) {
        report($e);

        return false;
    }
}
```

Ignoring Exceptions By Type

The `$dontReport` property of the exception handler contains an array of exception types that will not be logged. For example, exceptions resulting from 404 errors, as well as several other types of errors, are not written to your log files. You may add other exception types to this array as needed:

```php
/**
 * A list of the exception types that should not be reported.
 *
 * @var array
 */
protected $dontReport = [
    \Illuminate\Auth\AuthenticationException::class,
    \Illuminate\Auth\Access\AuthorizationException::class,
    \Symfony\Component\HttpKernel\Exception\HttpException::class,
    \Illuminate\Database\Eloquent\ModelNotFoundException::class,
    \Illuminate\Validation\ValidationException::class,
];
```

The Render Method

The `render` method is responsible for converting a given exception into an HTTP response that should be sent back to the browser. By default, the exception is passed to the base class which generates a response for you. However, you are free to check the exception type or return your own custom response:

```php
/**
 * Render an exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $exception
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $exception)
{
    if ($exception instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

    return parent::render($request, $exception);
}
```

Reportable & Renderable Exceptions

Instead of type-checking exceptions in the exception handler's `report` and `render` methods, you may define `report` and `render` methods directly on your custom exception. When these methods exist, they will be called automatically by the framework:

```php
<?php

namespace App\Exceptions;
```

```php
use Exception;

class RenderException extends Exception
{
    /**
     * Report the exception.
     *
     * @return void
     */
    public function report()
    {
        //
    }

    /**
     * Render the exception into an HTTP response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function render($request)
    {
        return response(...);
    }
}
```

You may type-hint any required dependencies of the `report` method and they will automatically be injected into the method by Laravel's service container.

HTTP Exceptions

Some exceptions describe HTTP error codes from the server. For example, this may be a "page not found" error (404), an "unauthorized error" (401) or even a developer generated 500 error. In order to generate such a response from anywhere in your application, you may use the `abort` helper:

```php
abort(404);
```

The `abort` helper will immediately raise an exception which will be rendered by the exception handler. Optionally, you may provide the response text:

```php
abort(403, 'Unauthorized action.');
```

Custom HTTP Error Pages

Laravel makes it easy to display custom error pages for various HTTP status codes. For example, if you wish to customize the error page for 404 HTTP status codes, create a `resources/views/errors/404.blade.php`. This file will be served on all 404 errors generated by your application. The views within this directory should be named to match the HTTP status code they correspond to. The `HttpException` instance raised by the `abort` function will be passed to the view as an `$exception` variable:

```
<h2>{{ $exception->getMessage() }}</h2>
```

You may publish Laravel's error page templates using the `vendor:publish` Artisan command. Once the templates have been published, you may customize them to your liking:

```
php artisan vendor:publish --tag=laravel-errors
```

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
- Scout
- Socialite

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.