Enlighten Documentation

Release 0.1.0-pre

Roy de Jong

Sep 27, 2017

Contents

CHAPTER 1

Quickstart

This guide will help you set up Enlighten and start using its powerful tools.

Installation

The Enlighten framework is available as a composer library. It couldn't be easier to install. Just set up your project, initialize Composer, and require the framework as a dependency:

```
composer require enlighten/framework
```

Your project is now ready to use Enlighten. Not bad, right?

Tip: Not interested in letting Enlighten handle your entire application flow? The rest of this guide is not for you. Refer to the documentation for the individual components you'd like to use instead.

Configuring your web server

Enlighten is designed to handle all incoming HTTP requests itself and route them to the appropriate code. To accomplish this, you will need to configure your web server to direct all requests for your web application to a primary entry point, like `index.php`.

Example configuration for nginx:

```
server {
    ## Basic configuration
    listen 80;
    root /var/www/myapp/public;
    index index.php;
    server_name dev.myapp.com;
```

```
    ## Restrict all directory listings
    autoindex off;

    ## Set the error page to index.php. As index.php applies routing
    ## (based on REQUEST_URI), our own error page will show up.
    error_page 404 = /index.php;

    ## Rewrite everything to index.php, but maintain query string
    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    ## Proxy requests to php-fpm listening on a Unix socket
    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
        include fastcgi.conf;
    }
}
```

Example configuration for Apache:

```
RewriteEngine on

RewriteCond %{REQUEST_FILENAME} !index.php
RewriteRule .* index.php?url=$0 [QSA,L]
```

Set up your application

The `Enlighten` class acts as the heart of your *Application*. It ties all the framework's components together into one easy to use package: from request processing and routing to sending a response back to the user.

To get started, you will want to initialize the composer autoloader and initialize a new instance of `Enlighten`. Here's what a typical `index.php` might look like:

```
<?php

use Enlighten\Enlighten;

include '../vendor/autoload.php';

$app = new Enlighten();
$app->start();
```

This snippet should now print out a "Welcome to Enlighten" page if everything was set up correctly. Let's expand that code to add our first route now:

```
<?php

$app = new Enlighten();

$app->get('/', function () {
    echo 'Hello world!';
});

$app->start();
```

This snippet of code will simply print out the text `Hello world!` when you visit the root page of your application.

To summarize, here's what we've done so far:

- Initialize composer's autoloader, which will make our `use` statement work.
- Initialize a new application instance (`new Enlighten()`) with a blank configuration.
- Register a new **Route** for all `GET` requests sent to `/`, with a function.
- Start the application: parse the incoming request, route it to our function, and send a response back.

All that in just a few lines of code. And this is just a basic example: we have many more power tools at our disposal to do more cool stuff.

Router configuration

A *route* tells your application what code an incoming request should lead to: the path from HTTP request to application code. A basic route always consists of a **pattern** (what to match) and a **target function** (what should this route do when matched?).

Request methods

The `$app->get($pattern)` function we used in the example above registers a new route that only applies to `GET` requests. There are appropriate functions for all other common request methods as well, such as `$app->post($pattern)`.

If you'd like to register a route that applies to all request methods, you can use the `route` function instead:

```
<?php

$app->route('/', function () {
    // This function will be called irregardless of request method (GET, POST, etc)
    echo "Hello world";
});
```

Routing patterns

When you register a new route, you have to define a **pattern**. This is what incoming requests are matched against. There are a few cool things you can do with these patterns.

You can use Regex patterns for a bit more flexibility:

```
<?php

$app->route('/(index|home)(/?)', function () {
    // Matches "/index" or "/home", with an optional trailing slash
});
```

You can also define dynamic variables in your route definitions which you can then retrieve in your functions:

```
<?php

$app->get('/users/view/$id', function ($id) {
    echo "You asked to GET a user with ID $id";
});
```

Use the Context

There's a lot of power hidden under the hood in the form of the application context. For example, you can manipulate the entire request or read out posted data.

Enlighten supplies this information to the target functions in your routes by using *dependency injection*.

```
<?php

$app->get('/hello/$name', function (Request $request, $name, Response $response) {
    // Read a posted value
    $age = intval($request->getPost('age', 18));

    // Manipulate the response code
    $this->response->setResponseCode(ResponseCode::HTTP_IM_A_TEAPOT);

    // Say hello to the user
    echo "Hi there, $name. You are $age years old.";
});
```

Tip: We will inject the appropriate variables based on the parameters you define in your function. The order doesn't matter. For route variables, make sure the name matches. For other variables, make sure the type is correct. If we can't resolve a variable, a `NULL` value will be passed.

Using filters

You can apply filters to handle common tasks like authentication, logging and error handling.

For example, you could log every request or add a snazzy header:

```
<?php

$app->before(function (Request $request) {
    ExampleLogger::writeLog('User requested: ' . $request->getRequestUri());
});

$app->after(function (Response $response) {
    $response->setHeader('X-Powered-By', 'MyAwesomeApp/v1.0');
});
```

CHAPTER 2

Application

Enlighten app

The easiest way to get started with the framework is to use the **Enlighten** class. This class represents the core of your application and ties all the framework components together in one easy to use package.

The flow for setting up an Enlighten application is as follows:

1. Initialize a new Enlighten application: `new Enlighten();`.
2. Configure your application as you please: register filters, routes and settings.
3. Call the `start()` function on your application instance.

A typical `index.php` might look like this:

```
<?php

use Enlighten\Enlighten;

include '../vendor/autoload.php';

$app = new Enlighten();
// (Register routes and settings here)
$app->start();
```

The `start()` function begins processing the incoming request, and will ultimately send a response back to the user.

Application flow

A typical application flow will like this:

1. The user submits an incoming HTTP request to your web server (e.g. nginx or apache).

2. Your web server proxies all requests (to non-static files) through to your `index.php` file. This file then initializes and configures your application.
3. Enlighten parses the incoming HTTP request and passes it through to the *Router* (when you call `$app->start()`).
4. If the request matches a route, that route's target function will be called (for example, a Closure or a function within a controller).
5. A response is gradually built up by your application code, and possibly modified by your filters. At the end of the run, the request is sent back to the user.

If you do not use the `Enlighten` class, this flow probably still applies to you; you'll just have to do a bit more work to tie all the components together.

Configuration

The `Enlighten` class offers a few helpful utilities for configuring your application. You can call these functions at any time before you call the *start()* function.

Overrides

By default, Enlighten does some of the initialization work for you. You can override certain components as needed.

- You can set a custom *Router* class by using the `setRouter($router)` function.
- You can set a custom HTTP request object by using the `setRequest($request)` function.

Routing

Tip: For more information on setting up and configuring routes, check out the *main article*.

You can either define and manage a custom router, or use the utilities in the `Enlighten` class to set up routing. You can either use the `$app->route($pattern, $target)` function to set up a route that matches requests of any method; or use a specific function like, for example, `$app->post($pattern, $target)` to set up a route for the POST request method only.

```
<?php

$app = new Enlighten();

// Create a route that matches all request methods
$route = $app->route('/my/page', function () {
    echo "Welcome to my route!";
});

// And register a filter on it, while we're here!
$route->after(function () {
    echo "And see you later!";
});

// Create a route specifically for the POST request method
$app->post('/my/form', function () {
    echo "Thanks for the post!";
});

// There's a function for each request method! e.g. post(), get(), put(), delete(), ..
↪.
```

You can set up a subdirectory for your application here as well:

```
<?php

$app = new Enlighten();
$app->setSubdirectory('/projects/myapp');
```

Filters

Tip: For more information on setting up and configuring filters, check out the main article.

You can hook in to certain application events by attaching filter functions to them. These filter functions have access to the Context (see below).

- `before()`: Called after the application has initialized, but before matching the request.
- `after()`: Called after routing has finished, and before the request is sent. Skipped when an exception occurs.
- `onException()`: Called if an Exception is raised during application execution.
- `notFound()`: Found when no suitable route can be found. Note that both `before()` and `after()` are still called as well.

If a filter function explicitly returns `false`, then no other filter functions of the same type will be executed after that. In the case of the `before` filter, this will also prevent the application from continuing.

```
<?php

$app = new Enlighten();
$app->notFound(function (Request $request) {
    echo "Sorry, but that page is not here: " . $request->getRequestUri();
});
```

You can also apply filter functions to specific routes rather than the application scope. Check out the *routing docs* for details.

Context

The application context (class `Context`) is a collection of data that represents the current state of the application. At its core, it is simply a bag of objects that is filled up and passed around the application, constantly being fed with the most up-to-date information.

The magic comes from its ability to intelligently inject its contents into a function based on its parameter list. For example, if the context contains a *Request* object, and a function requests that type of data in its parameter list, it can be passed as a value to that function. This is used throughout the framework as a way to flexibly pass data on-demand without making your code more verbose.

The `Enlighten` class normally initializes and manages its own Context, which you cannot directly modify at configuration time. If you manage your own application flow, you'll need to set up your own context to pass around. Because the *Context* class is completely generic you could also apply it to a variety of other uses.

```
<?php

$app->get('/hello/$name', function (Request $request, $name, Response $response) {
    // Read a posted value
```

```
    $age = intval($request->getPost('age', 18));

    // Manipulate the response code
    $this->response->setResponseCode(ResponseCode::HTTP_IM_A_TEAPOT);

    // Say hello to the user
    echo "Hi there, $name. You are $age years old.";
});
```

We will inject the appropriate variables based on the parameters you define in your function. The order doesn't matter. For *routing* variables, make sure the name matches. For other variables, make sure the type is correct. If we can't resolve a variable, a `NULL` value will be passed.

What does the Context contain?

The `Enlighten` class will always publish the following data to the context it manages:

- **Enlighten**: The application instance itself.
- **Request**: The parsed incoming HTTP request object.
- **Response**: The HTTP response that is being built up.
- **Router**: The router managed by the application.
- **Route**: The route that matched - if one was successfully matched.
- **Exception**: The last exception that was raised if there was one - particularly useful for `onException()` filters.
- **Context**: The `Context` itself: use this reference to inject your own variables.

Managing a context

```
<?php

$context = new Context();

// Register a Request object to our Context
$request = new Request();
$context->registerInstance($context);
```

When you register an object to a context, it will override any previous registrations of the same type. In the example above, if we had previously set a `Request` on the context, it has now been replaced.

If the object that you have registered has parent classes, *weak links* will be created for those classes as well. That means if you register an `InvalidArgumentException`, a weak link will also be created for `Exception`. So when a function asks for an `Exception`, they may still get the most recent compatible object instead - an `InvalidArgumentException` in this example. However, if an exact match is available in the Context, that object will always be used instead.

Note that you can currently only register *objects* to a context; primitive types by variable name are not supported at this time.

Manual injection

When you have a context, you will also want to use its superpowers to inject its values to a function. All you need is a *callable* function with a parameter list.

```
<?php

// Set up our context
```

```
$context = new Context();
$context->registerInstance($request);
$context->registerInstance($response);

// Let's define our Closure that will receive dependency injection.
$sampleFunction = function (Request $request, $randomVar, Response $response) {
    // ...
});

// Use the context to determine parameters, and call the function
$params = $context->determineParamValues($sampleFunction);
call_user_func_array($sampleFunction, $params);
```

This is the only way you can currently extract values from a context.

Quirks

Here is an overview of quirks that you may need to know about when using the `Enlighten` class:

- If any output is sent after `start()` is called and before the HTTP response is sent back, it will be appended to the end of the response body "just in time". That means you can use *echo* freely throughout your code, but note your output will appear at the very end of the response.
- If an error occurs (including 404 errors), the output buffer is cleared and the response is emptied. Any output sent by your filter functions, for example, will be discarded.
- `after()` filters have the final say on any output that is sent out - their output is never discarded. But: they will not be called if an exception occurs in your application.
- If your *Router* is empty, a default "Welcome to Enlighten" page will be shown.

CHAPTER 3

Router

A router manages a collection of routes. Each route maps an incoming request to an appropriate target function. It is an essential component that lets your application respond to user's requests intelligently.

Creating a router

Tip: Under normal circumstances, the `Enlighten` *Application* class will initialize and manage a default router for you - you do not necessarily need to initialize or manage one yourself.

If you want more control over the router and its routes, or use a custom implementation of the `Router` class, you will need to manage it yourself:

```php
<?php

// Initialize a custom router
$router = new Router();

// Assign our router to our application
$app = new Enlighten();
$app->setRouter($router);
$app->start();
```

Configuring routes

You can define a route by simply intializing it. A basic route always consists of a **pattern** (what to match) and a **target function** (what should this route do when matched?):

```php
<?php
```

```
// Create our route
$route = new Route('/users/$user/$action', function ($user, $action) {
    // Will match /users/admin/view
});

// Register it to our router
$router->register($route);
```

When you use the `Enlighten` class - either with the default router or after registering your own router - you can also use the `route()` function. This function also returns the created *Route* object so you can customize it further as needed - even without ever having to manage your own Router.

```
<?php

$app->route('/users/$user/$action', function ($user, $action) {
    // ...
});
```

In addition, the `Enlighten` class also offers some utility functions that let you register routes with request method constraints. There are utility functions for the `GET`, `POST`, `PUT`, `PATCH`, `OPTIONS` and `DELETE` methods. The `route()` function does not apply such a constraint, so it will match any method by default.

```
<?php

// For example, register a route with a GET request method constraint:
$app->get('/sample', ...);

// Or a DELETE constraint:
$app->delete('/sample', ...);
```

You can clear a router by calling `$router->clear()`. You can assert whether any routes have been registered in a router via the `$router->isEmpty()` function.

Routing patterns

The primary factor for any route is its pattern. A pattern indicates what the incoming Request's URL needs to be matched against for the route to be considered a successful match.

Pattern matching always occurs in groups (virtual URL levels separated by the */*). All given pattern groups must be present for it to match. You can also use some simple Regex patterns in your route pattern.

URL variables

In addition to defining static routing patterns that need to match exactly, you can also make them dynamic by using URL variables.

You can use dynamic variables in your routing pattern by using the dollar sign *$* as a prefix. A variable must always span one entire URL group. */bla/$var* is okay but */bla$var* will not work as expected.

Dynamic variable values will be passed back to the target function via dependency injection:

```
<?php

$app->route('/say/$string', function ($string) {
```

```
    // By defining $string as a parameter for your target function, the value of
↪$string will be set to the corresponding URL group that was in the request URI that␣
↪matched.
});
```

It is not possible to apply any particular constraints to what is accepted as a value for a URL variable, so always make sure to carefully validate all values that are supplied by the user.

A URL variable does not make that part of the pattern optional.

Target functions

A target function must either be **callable** or a **function definition string**. Here's an overview of the most common ways this is accomplished:

```
<?php

// 1. Use a Closure function
$app->route('/example', function () { });

// 2. Use an Enlighten function definition string
$app->route('/example', 'my\namespace\Class@myFunction');
// Format: classNameWithNamespace@functionName

// 3. Use a static class
$app->route('/example', ['MyClassName', 'myFunctionName']);
$app->route('/example', 'MyClassName::myFunctionName');

// 4. Use an object function
$app->route('/example', [$myClassObj, 'myFunctionName']);
```

Target functions have access to the application context and receive dependency injection - this allows them to retrieve certain objects on demand.

Tip: This section only highlights some common techniques - PHP.net has more examples on other ways to define callable functions.

Using subdirectories

The Router class supports operating out of a subdirectory. This can be useful if you want to run your entire application from a certain directory or with a certain prefix.

```
<?php

// Either directly via a custom router
$router->setSubdirectory('example');

// ..or using the Enlighten class
$app->setSubdirectory('example');
```

If you follow the above example, the router will assume that all your routes will begin with a "example" directory. For example, if you register a route for `/mypage` it will then only match against requests for `/example/mypage`.

Redirects

You can quickly add temporary or permanent redirects using the `addRedirect()` function. Internally, this will create and register a route that performs a redirection.

```
<?php

$router->createRedirect('/from', '/to', $permanent = false);

// Add a permanent redirect to another page
$router->createRedirect('/page/old', '/page/new', true);

// Add a temporary redirect to an external site
$router->createRedirect('/jfgi', 'http://www.google.com');
```

You can use variables in the redirect pattern, but these variables are currently only used for matching and cannot be utilized. If you have more complex redirect requirements, we suggest adding a redirect route manually.

The `Enlighten` class also offers a convenience function to register redirect routes:

```
<?php

$app->redirect('/from', '/to', $permanent = false);
```

CHAPTER 4

Request

The `Request` class lets you parse and examine the HTTP request that the user sent to your web server.

Tip: Instead of accessing PHP superglobals like `$_POST`, `$_GET`, `$_SERVER`, `$_COOKIES`, etc. you should always use the Request object – it is a safer, more consistent and more convenient way of dealing with user-submitted data.

Accessing Request

The `Request` object is made available through the Application Context. You can access it through your filter functions and target functions by adding it to your parameter list. Enlighten will inject the dependency where it is needed.

```
<?php

new Route('/', function (Request $request) {
    echo $request->getRequestUri();
});
```

By default, a request object is generated when you use the `Enlighten` class based on the current PHP environment and superglobals. You can also initialize and set a custom request by calling `$app->setRequest()`. If you do not use the `Enlighten` class you can parse your own request by calling `Request::extractFromEnvironment()`.

Reading data

Users can submit data to your application in a variety of ways. For example, they may use a `POST` request to submit a form. Or they may use a `GET` request and supply query string parameters. Here's a summary of the options, and what they are called in Enlighten:

- Posted values: User-submitted data in `POST`, `PUT` or `PATCH` requests. Called `$_POST` in vanilla PHP.
- Query string parameters: Parameters added in the Request URL by the user - in any request type. Misleadingly called `$_GET` in vanilla PHP.
- Cookies: Little bits of key/value data that are stored on the client computer. Called `$_COOKIE` in vanilla PHP.
- Uploads: Files that the user uploaded as part of their request. Called `$_FILES` in vanilla PHP.
- Headers: A variety of request headers that typically contain technical data. Hidden away in `$_SERVER` in vanilla PHP.

Reading posted values and query parameters

- You can read a single value by calling `getPost($key)` or `getQueryParam($key)`. This function will try to look up the appropriate value and return a *string*. If your value cannot be located, `NULL` will be returned.
- You can also supply a default value when that will be returned instead of `NULL` when your given `$key` cannot be located: use `getPost($key, $defaultValue)` or `getQueryParam($key, $defaultValue)`.
- You can get a full key/value array of all raw values by calling `getPostData()` or `getQueryParams()`.

```php
<?php

// Read a single value. Will return NULL if the "name" key is not found.
$name = $request->getPost('name');

// Read a single post value. Will return 18 if the "age" key is not found.
$age = intval($request->getPost('age', 18));

// Retrieve a key => value array of all POST values.
$postArray = $request->getPostData();

foreach ($postArray as $key => $value) {
    // ...
}
```

Working with URLs

When a user submits a request to your web server, they do so for a specific hostname (for example, the name or IP address of your website) and a request URL (the page they wish to access on your website - including query string parameters).

Typically, when you configure your web server you will let it determine which website to serve based on its hostname. Your application then simply interprets the request URL (everything starting at the `/`). The *Routing* module helps you set this up - it maps an incoming request to a piece of code in your application.

The Request class offers you some utilities for reading out this data:

- `getProtocol()`: Gets the protocol string, either "http" or "https", that was used to access your website. You can also use `isHttps()` to determine whether HTTPS was used.
- `getHostname()`: Gets the requested hostname (e.g. "google.com").
- `getPort()`: Gets the port number that was used to talk to your web server (e.g. 80, 443 or a non-standard port like 8080).
- `getRequestUri($includeQueryString = false)`: Gets relative request URL (e.g. "/my/page.html") and, optionally, query string.

- `getUrl($includeQueryString = true)`: Gets the full current URL - including protocol, hostname, port (if it is non-standard), request URL and, optionally, query string.

CHAPTER 5

Code Examples

“Hello world” application

A simple `index.php` script that will simply set up *an Enlighten application* and print out “Hello, World!” when opening the root page.

```
<?php

use Enlighten\Enlighten;

include '../vendor/autoload.php';

$app = new Enlighten();

// Say "Hello, World!" for all GET requests to "/"
$app->get('/', function () {
    echo "Hello, World!";
});

$app->start();
```

Note that you’ll need to configure your server to forward all requests to this script first.

Forms

This code shows you how you can deal with form submissions.

```
<?php

$app->get('/', function () {
    // View logic goes here: display your form
});
```

```php
$app->post('/', function (Request $request) {
    $name = $request->getPost('name', 'John Doe');
    echo "Hi there, $name";
});
```

Custom error pages

The framework offers some basic error pages by default, but you can override them by using filters on your application object.

```php
<?php

// Generic error handler
$app->onException(function (\Exception $ex) {
    echo "Sorry, something went wrong!";
    echo $ex->getMessage();
});

// 404 / route not found error handler
$app->notFound(function (Request $request) {
    echo "Sorry, but the page you requested could not be found!";
    echo "You requested: " . $request->getRequestUri();
});
```

If one of your filter functions causes any output to the body, the framework's default error pages will be suppressed.

Read and set cookies

The **Request** and **Response** objects can be used for reading and writing cookies, respectively.

```php
<?php

$app->get('/', function (Request $request, Response $response) {
    // Iterate all cookies
    $cookies = $request->getCookies();

    foreach ($cookies as $name => $value) {
        echo "Cookie $name = $value" . PHP_EOL;
    }

    // Set some cookies (follows same format as php set_cookie)
    $response->setCookie('SomeCookie', 'SomeValue', time() + 60, '/');
});
```

Handle file uploads

The **Request** class has an easy to use facility for safely processing file uploads.

```php
<?php

$app->post('/', function (Request $request) {
    // Iterate all uploaded files
    $files = $request->getFileUploads();

    foreach ($files as $file) {
        // Is this file okay?
        if ($file->hasError()) {
            echo $file->getErrorMessage();
            continue;
        }

        // Let's move it to our uploads directory
        $filename = uniqid() . '.tmp';
        $file->saveTo("./uploads/$filename");
    }
});
```

You can call the `saveTo()` function multiple times if you want more than one copy of a file.

It's a good idea to always generate your own file names, as the user-supplied filename (`$file->getOriginalName()`) is not necessarily safe to use, and is not unique either.

CHAPTER 6

What is Enlighten?

Enlighten is a micro framework that helps you rapidly build PHP web applications.

Take the pain out of the common tasks you have to go through when you build any PHP application:

```
$app = new Enlighten();

$app->get('/hello/$name', function ($name) {
    echo "Hi there, $name";
});
```

In short: a fat-free framework with no external dependencies that simply helps you *get shit done* without getting in your way:

- Deal with parsing requests with ease: cookies, file uploads, headers, forms and more.
- Set up flexible and easy to use routing for requests, with dynamic URL variables and dependency injection.
- Apply filter functions for routing events and error handling; great for common tasks like authentication and logging.
- Control every aspect of the HTTP response sent back to the user, with an easy to use API.

Enlighten is an open source, MIT licensed project. Check it out on GitHub:

https://github.com/roydejong/Enlighten

CHAPTER 7

The documentation

You can always find the latest documentation on *readthedocs.org*:

https://enlighten.readthedocs.org/en/latest/

Tip: Want to see why Enlighten is awesome, and start building cool stuff right away? Check out the aptly named *Quickstart* section for the quick and dirty code examples.

CHAPTER 8

Support

If you have found an issue, have an idea, or want to ask a question, please do so by creating a GitHub ticket:
https://github.com/roydejong/Enlighten/issues