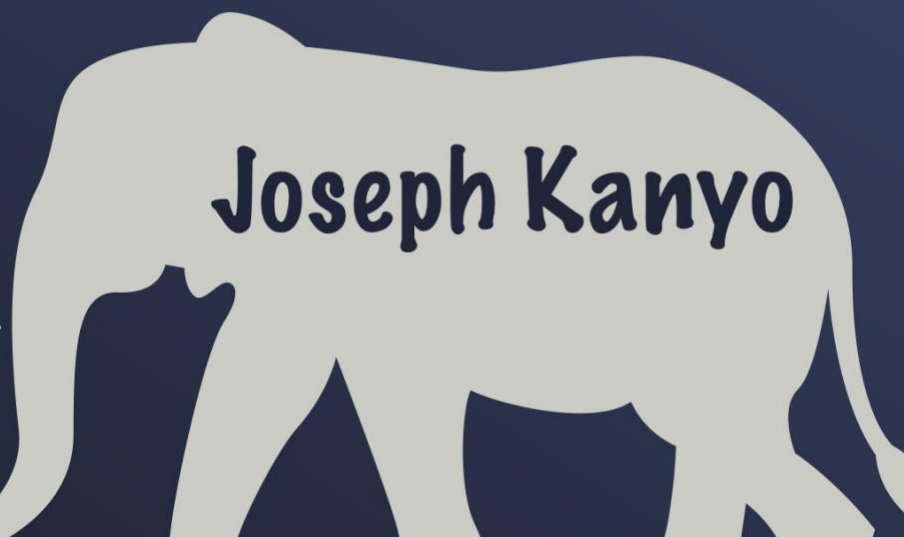
Senior

???

Medior PHP
The Missing Step

Junior

Intern



Joseph Kanyo

Configuration Object

Now we have our **env variables** loaded and we added some validation to them, but we access them directly, using the `$_ENV` superglobal, which is not a good practice. We can't add additional logic to convert env variables from string to integers or booleans and we want to inject these variables to different constructors using our **DI Container** and not direct access.

The solution is really simple, we just use a `Configuration` object: there are lots of libraries for this, or you can just develop your own one:

File: ./src/Configuration.php

```php
<?php

declare(strict_types=1);

namespace KanyJoz\CodeFlash;

readonly class Configuration
{
    public function __construct(private array $configs) {}

    // It gets a dot separated string key
    // Otherwise we return the default value if supplied,
    // if not we return null as default
    public function get(string $name, mixed $default = null): mixed
    {
        $path  = explode('.', $name);
        $value = $this->configs[array_shift($path)] ?? null;

        if ($value === null) {
            return $default;
        }

        foreach ($path  as $key) {
            if (! isset($value[$key])) {
                return $default;
            }

            $value = $value[$key];
        }

        return $value;
    }
}
```

Next we create an `app.php` file in the **config folder** to store the array that we create from the `$_ENV` superglobal, and this will be the array that we pass down to the `Configuration` object.

File: ./config/app.php

```php
<?php

declare(strict_types=1);

return [
    'app' => [
        'environment' => $_ENV['APP_ENV'],
        'name' => $_ENV['APP_NAME'],
    ],
];
```

File: ./config/path_constants.php

```php
<?php

declare(strict_types=1);

// root
const ROOT_PATH = __DIR__ . '/..';

// bin
const BIN_PATH = ROOT_PATH . '/bin';
const CONTAINER_PATH = BIN_PATH . '/container.php';
const DOTENV_PATH = BIN_PATH . '/dotenv.php';

// config
const CONFIG_PATH = ROOT_PATH . '/config';
const BINDINGS_PATH = CONFIG_PATH . '/bindings.php';
const DOTENV_RULES_PATH = CONFIG_PATH . '/dotenv_rules.php';
// Add new constant
const APP_CONFIG_PATH = CONFIG_PATH . '/app.php';

// public
const PUBLIC_PATH = ROOT_PATH . '/public';

// tmp
const TMP_PATH = ROOT_PATH . '/tmp';

// var
const VAR_PATH = ROOT_PATH . '/var';
```

Then we just add this object to our Container.

File: ./config/bindings.php

```php
<?php

declare(strict_types=1);

use KanyJoz\CodeFlash\Configuration;
use Psr\Container\ContainerInterface;
use Slim\App;
use Slim\Factory\AppFactory;

return [
    App::class => function(ContainerInterface $container) {
        return AppFactory::createFromContainer($container);
    },

    // We return a new Configuration object and pass it the app.php array
    Configuration::class => function() {
        return new Configuration(require_once APP_CONFIG_PATH);
    },
];
```

Middlewares

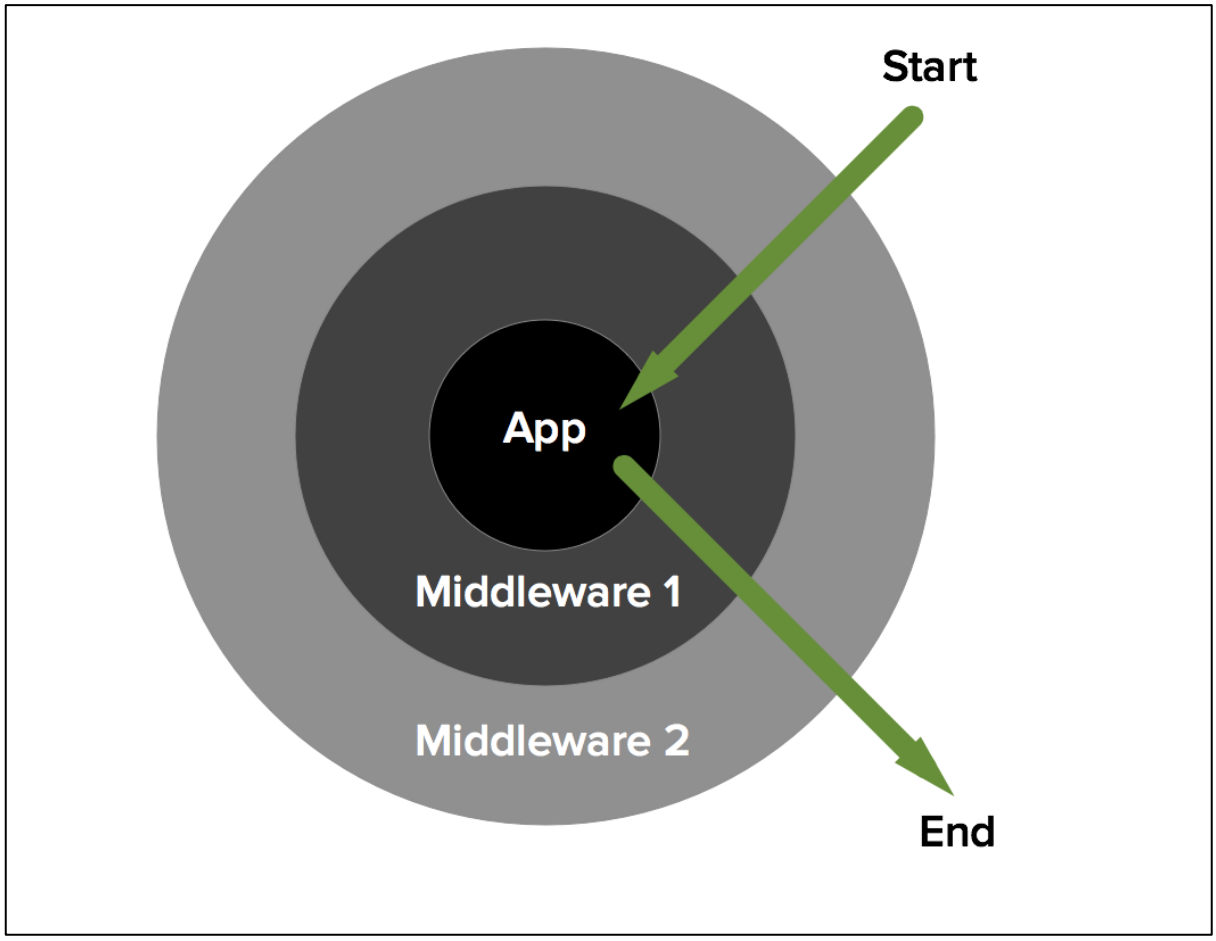
When you're building a web application there's probably some shared functionality that you want to use for many (or even all) HTTP requests. For example, you might want to log every request, compress every response, or check a cache before passing the request to your controller actions.

A common way of organizing this shared functionality is to set it up as *middleware*. This is an additional building block to our framework. I decided to add it here, so that we can later create middlewares as necessary.

How Middlewares work?

The Slim documentation has a really good image about middlewares, let's take a look at it:

Source: https://www.slimframework.com/docs/v3/concepts/middleware.html



- The **Start** describes the point in time when the `Request` object was made from the **PHP Superglobals**
- **Th**e `App` object is built up and it has a **Stack (LIFO)** of `Middleware` objects
- The **lastly pushed in** `Middleware` is called first and it gets the `Request` object. It can modify it, do some logic and so on. It can either give the `Request` object to the next `Middleware` in the **Stack**, or it can return early with a `Response` object itself without us never hitting the `Router`

  If the `Request` traveled over all the `Middleware` objects of the **Stack**, it arrives at the `Router`, so basically this is where one of our **Controller** actions is called
- If the business logic is successful then the **Controller action** creates a `Response` object
- But before the `Response` object travels back to the client, it goes through all of the `Middleware` objects **in reverse order**: the last `Middleware` that got the `Request` will be the first to get the `Response`
- And after all this is done, then the `Response` is sent to the client

PSR Tea Time

You migth have not noticed, but let's just take a quick break to list how many PSR standards did we already made use of:

- We are following **PSR-1** and **PSR-12** standards written about **coding style**
- We used **PSR-4** actually, because the `autoload.php` that composer gives us implements PSR-4
- We actually required and used **PSR-7** interfaces to handle **HTTP traffic**
- We actually also have the ability to use the **PSR-17 HTTP Factory** interfaces to produce our own `Request` or `Response` objects (will do so in a later chapter)
- We are using **PHP-DI** which follows the `ContainerInterface` described by **PSR-11**
- And now we are talking about middlewares that are implemented by Slim using the **PSR-15** interfaces: `RequestHandlerInterface` and `MiddlewareInterface`

Local and Global Middlewares

You actually have the ability to configure your `Middleware` objects to a sepcific route as well. Most of the times you want to have a `Middleware` logic that is executed for every HTTP request, but sometimes you only want to execute it for one or for a group of routes.

This is really easy in **Slim** and we will see in the next chapter how to do it.

Lazy and Eager Middlewares

You will see that you can register `Middleware` objects to your routes in 2 ways in **Slim**. You can call the `addMiddleware()` function on a route and pass an initialized middleware object: `addMiddleware(new MyMiddleware()).` Or you can call the `add()` function and pass the name of the class, for example `add(MyMiddleware::class)`

The first solution is **eager**: when the HTTP request comes in and the `App` object is built up to handle it, the **middleware stack** is also built and then it will create this object in memory.

The second option is lazy: meaning that when the `App` is built it will only save the name of the class, and then resolve the `Middleware` object from the **DI Container** when it is actually needed.

I recommend the **lazy way**, the container will make less mistakes than us, and there are situations when your earlier middlewares on the chain will return early with a `Response` object, then you built a `Middleware` object in the memory that is not going to be used, instead of storing only a string to its classname. The lazy way is less error prone as well, becuase sometimes it can happen that your other `Middleware` objects will setup dependencies for your later ones on the chain, meaning it is not enough for your app that they are built in the memory, the HTTP logic needs to be also executed. But not if you lazy build them.

Sessions (and Cookies)

The last big part we want to create is to have users logging in and out, so we want to store state. As you probably know **HTTP is a stateless protocoll**, so we can do so by using cookies and/or sessions.

Cookies are managed by the client (usually your bowser) and not safe to store sensitive information, so for users we definitely will need to set up sessions in our application.

So the upcoming chapters let's set it up and use it for **flash messages first**. If we fill in the form and don't put validation errors in there I want to read a nice message on the `show.twig` page, that a new card was created. And for flash messages in a traditional web application, typically we use sessions.

But before we go there let's just remember why did I put cookies in the title as well, if we are not even going to use them? Because sessions typically work with cookies, so before we start, let's refresh the steps on how PHP' session management works:

- We need to call the built-in `session_start()` function, and we want to actually call it on every HTTP request, before we reach the controller action. The best approach is to write a middleware for this, or most often good session packages come with their own middleware
- The `session_start()` call will then check the incoming `Request` headers and see if there is a cookie in the **Cookie: header** called PHPSESSID.
- When we first arrive on the page it will not be there, so the `session_start()` will set a `Response` **header Set-Cookie:** with the PHPSESSID and its value
- This `Response` headers is understood by your browser and it will save this PHPSESSID cookie into a file somewhere on your host
- And on every subsequent request your browser will automatically put Cookie `Request` header into the headers passing this cookie
- So this time `session_start()` will find it, then grabs the value from the PHPSESSID cookie, which is a unique hash belonging to the user
- Then by default the PHP session mechanism stores data for each of the session ids in the filesystem of the webserver (you can modify this in your `php.ini` file with `session.save_path` option)
- It grabs the data for this id and puts it into the `$_SESSION` superglobal so we can use it

- And when we send the `Response`, but we put new data into this user's session, it will override it with the new data

One additional thing you can do is to set up a table in your database and save the session data there. There are packages for this, but you can build your own solution.

Also most often it is done with redis instead of a relational database. By saving the session data to redis we get the data from the memory instead of the filesystem of the webserver and we also separate this functionality from the webserver.