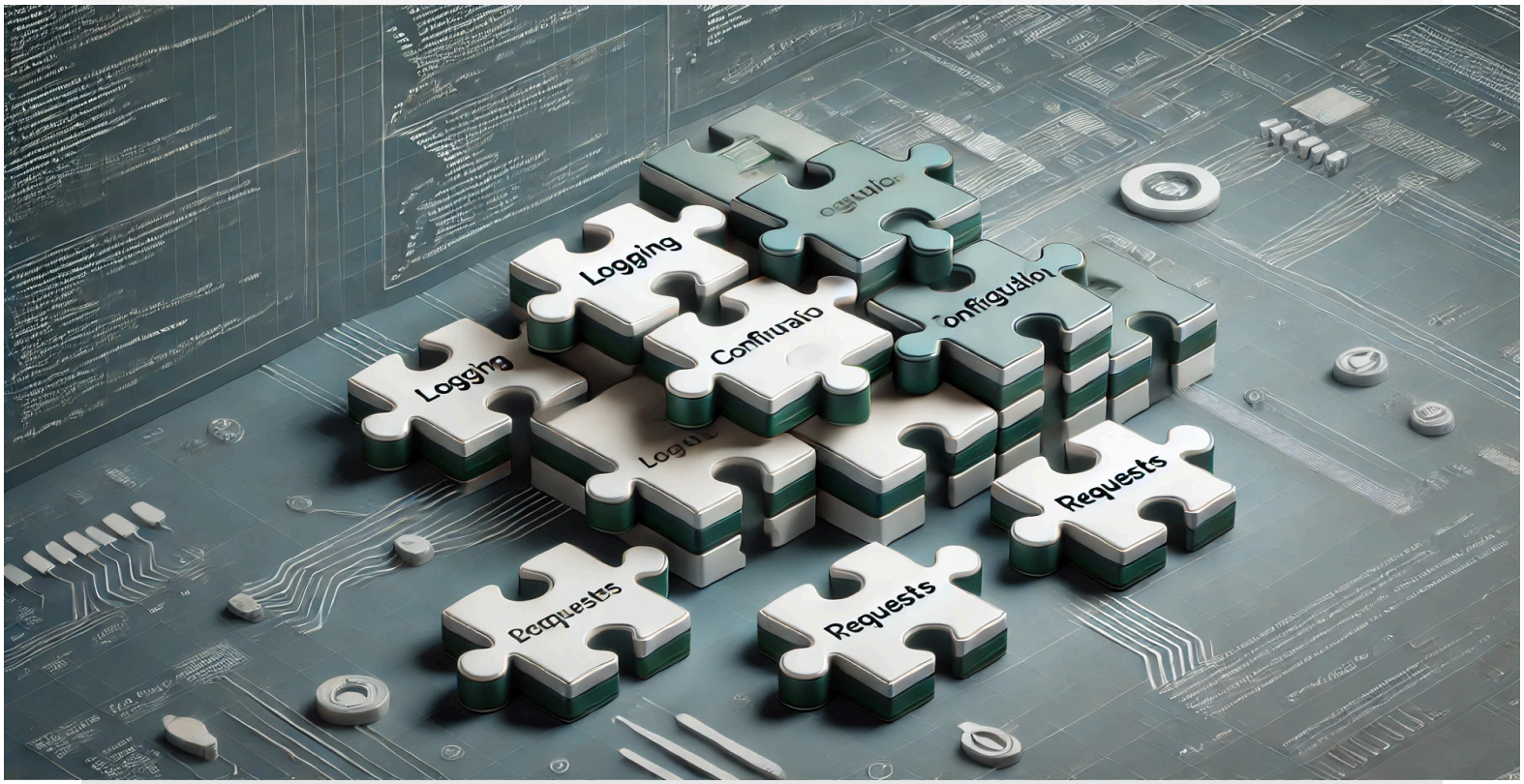
TABLE OF CONTENTS

POURQUOI UNE APPROCHE MODULAIRE ? 4
DÉFINIR NOTRE MODULE PRINCIPAL (“CORE”) 4
CLASSE BOOTSTRAP 4
CLASSE DE SECOURS (FALLBACK MODULE) 7
CLASSE DE CONFIGURATION (REMPLACEMENT DE “CONFIGURATOR”) 8
CLASSE LOGGER (CORELOGGER) 10
CLASSE REQUEST 13
CONCLUSION 15
MOTS-CLÉSGENERALCORE-FRAMEWORK-FRFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN 15



PARLONS-EN – CONSTRUIRE UN FRAMEWORK PHP MODULAIRE DE A À Z

Auteur(s) : Louis Ouellet —-

Avez-vous déjà travaillé avec des frameworks PHP populaires comme CakePHP ou Symfony et pensé : « Comment ont-ils été conçus ? » Les frameworks PHP peuvent être de formidables économies de temps, mais ils révèlent leur plein potentiel seulement quand on les maîtrise en profondeur. Développer votre propre mini-framework est un excellent exercice d'apprentissage, car cela offre une meilleure compréhension des bonnes pratiques, de la modularité et de la maintenabilité.

Dans cet article, je partage comment j'ai commencé à bâtir mon propre framework PHP depuis zéro. Cela inclut la création d'une classe Bootstrap, la gestion de la configuration, la mise en place de modules, la création d'un système de journalisation (logging) et enfin l'encapsulation des requêtes dans une classe Request. En explorant chaque morceau, vous découvrirez l'intérêt d'une approche modulaire et bien structurée, pouvant être étendue grâce à des modules personnalisés.

POURQUOI UNE APPROCHE MODULAIRE ?

La modularité est essentielle pour créer des applications légères et flexibles. Un framework volumineux peut ajouter trop de surcoût si vous n'avez besoin que d'une ou deux fonctionnalités spécifiques. En scindant les fonctionnalités en plusieurs modules chargés au besoin, on assure à la fois performance et maintenabilité.

DÉFINIR NOTRE MODULE PRINCIPAL ("CORE")

J'ai décidé de nommer le module principal simplement `core`. Il inclut les classes de base dont les autres modules pourraient avoir besoin, ainsi que la logique essentielle pour initialiser l'application. Un point important : nous voulons que chaque module ne s'initialise qu'une seule fois.

CLASSE BOOTSTRAP

La classe `Bootstrap` détermine quels modules sont chargés, en fonction du « scope » de l'application (par exemple : Router, CLI, API). Voici le code en cours de développement. Notez que nous utilisons un tableau de configuration par défaut et que nous autorisons des substitutions via notre système de configuration (illustré plus tard) :

Source: Bootstrap.php

src/Bootstrap.php

```php
<?php

/**
 * Core Framework - Bootstrap
 *
 * @license    MIT (https://mit-license.org/)
```

```
 * @author     Louis Ouellet <louis@laswitchtech.com>
 */

namespace LaswitchTech\Core;

use LaswitchTech\Core\Configurator; // Soon replaced with Config
use LaswitchTech\Core\Module;
use Exception;

class Bootstrap {

    const Default = [
        "LOGGER" => [
            "class" => "\LaswitchTech\coreLogger\Logger",
            "scope" => [
                "Router",
                "API",
                "CLI"
            ]
        ],
        ...
    ];

    /**
     * Constructor.
     */
    public function __construct($scope){

        // Set the global variable
        global $CONFIGURATOR;

        // Initialize Configurator (we'll replace this with our new
Config class)
        $CONFIGURATOR = new Configurator(['bootstrap']);

        // Retrieve the straps
        $straps = $CONFIGURATOR->get('bootstrap');

        // Loop through the straps
        foreach(self::Default as $strap => $config){

            // Check if an alternate strap exist
            if(isset($straps[$strap])){
```

```
                foreach($config as $key => $value){
                    if(isset($straps[$strap][$key])){
                        $config[$key] = $straps[$strap][$key];
                    }
                }
            }

            // Check if strap matches scope
            if(!in_array($scope, $config['scope'])) continue;

            // Initialize the Global Variable
            global ${$strap};

            // Set Class
            $class = $config['class'];

            // Check if the class exists
            if(class_exists($class)){
                // Initialize the class in the global namespace
                ${$strap} = new $class();
            } else {
                // Fall back to a default module class
                ${$strap} = new Module();
            }
        }
    }
}
```

Nous testons ensuite le résultat avec un petit script :

Source: test.php

test.php

```
<?php

use LaswitchTech\Core\Bootstrap;
require dirname(__DIR__) . "/vendor/autoload.php";
```

```php
$BOOTSTRAP = new Bootstrap("Router");

foreach(get_defined_vars() as $key => $value){
    // Skip php internal variables
    if(substr($key,0,1) == "_" || in_array($key,["argv","argc"]))
continue;
    echo "Variable Name: " . $key . " = " . get_class($value) . PHP_EOL;
}
```

CLASSE DE SECOURS (FALLBACK MODULE)

Un inconvénient du caractère optionnel des modules est qu'il faut vérifier si chacun a bien été chargé (c.-à-d. non nul). À la place, nous pouvons créer une classe par défaut `Module` qui signale au développeur lorsqu'on appelle une méthode dans un module non installé :

Source: Module.php

src/Module.php

```php
<?php

/**
 * Core Framework - Module
 *
 * @license    MIT (https://mit-license.org/)
 * @author     Louis Ouellet <louis@laswitchtech.com>
 */

namespace LaswitchTech\Core;

use Exception;

class Module {

    /**
     * Dynamically handle calls to missing methods.
     */
```

```
    public function __call($name, $arguments){
        $message = "Warning: Method " . $name . " is not available. The class/module is not installed." . PHP_EOL;
        echo $message;
        error_log($message);
        exit;
    }
}
```

Grâce à cela, l'exécution s'arrête proprement et enregistre une erreur si un développeur tente d'utiliser un module non chargé.

CLASSE DE CONFIGURATION (REMPLACEMENT DE "CONFIGURATOR")

Ensuite, examinons notre nouvelle classe `Config` (anciennement `Configurator`). Elle gère nos fichiers de configuration au format JSON :

Source: Config.php

src/Config.php

```
<?php

/**
 * Core Framework - Config
 *
 * @license    MIT (https://mit-license.org/)
 * @author     Louis Ouellet <louis@laswitchtech.com>
 */

namespace LaswitchTech\Core;

use ReflectionClass;
use Exception;
```

```php
class Config {

    const Extension = '.cfg';
    const ConfigDir = '/config';
    private $Files = [];
    private $Configurations = [];
    private $Path = null;

    public function __construct($File = null){
        ...
    }

    protected function isAssociative($array) {
        ...
    }

    public function add($File, $Path = null){
        ...
    }

    public function get($File, $Setting = null){
        ...
    }

    public function list($byFiles = false){
        ...
    }

    public function set($File, $Setting, $Value){
        ...
    }

    public function delete($File = null){
        ...
    }

    public function check($File){
        ...
    }

    public function root(){
        ...
```

```
    }

    public function path($className) {
        ...
    }
}
```

CLASSE LOGGER (CORELOGGER)

Ci-dessous se trouve une version simplifiée de la classe `Logger` qui utilise notre nouvelle classe `Config` pour gérer divers réglages (niveau de journalisation, rotation, etc.). Elle montre également comment chaîner plusieurs méthodes pour rendre la journalisation plus fluide :

Source: Log.php

src/Log.php

```
<?php

/**
 * Core Framework - Log
 *
 * @license    MIT (https://mit-license.org/)
 * @author     Louis Ouellet <louis@laswitchtech.com>
 */

namespace LaswitchTech\Core;

use DateTime;
use Exception;
use ReflectionClass;

class Log {

    // Constants for log levels
    const DEBUG_LABEL = 'DEBUG';
```

```
    ...

    private $Path = null;
    private $Levels = [];
    private $Level = 0;
    private $IP = false;
    private $Rotation = false;
    private $Files = [];
    private $File = null;
    private $Message = null;

    public function __construct(){
        global $CONFIG;
        $CONFIG->add('log');
        ...
    }

    public function config($level = null){
        ...
    }

    public function ip(){
        ...
    }

    public function agent(){
        ...
    }

    public function add($name, $path = null){
        ...
    }

    public function set($name){
        ...
    }

    public function clear($name = null){
        ...
    }

    public function read($name = null){
        ...
```

```
    }

    public function list(){
        ...
    }

    public function log($message, $level = self::LEVEL_INFO, $name =
null){
        ...
    }

    public function debug($message, $name = null){
        ...
    }

    public function info($message, $name = null){
        ...
    }

    public function success($message, $name = null){
        ...
    }

    public function warning($message, $name = null){
        ...
    }

    public function error($message, $name = null){
        ...
    }

    public function fatal(){
        ...
    }
}
```

CLASSE REQUEST

Enfin, nous avons la classe `Request`, qui encapsule les superglobales habituelles (`$_GET`, `$_POST`, etc.) dans une interface plus contrôlée. Nous les désactivons même dans l'espace global pour encourager l'utilisation exclusive de la classe. Cela permet un point central pour la désinfection ou la transformation des données.

Source: Request.php

src/Request.php

```php
<?php

/**
 * Core Framework - Request
 *
 * @license    MIT (https://mit-license.org/)
 * @author     Louis Ouellet <louis@laswitchtech.com>
 */

namespace LaswitchTech\Core;

use Exception;

class Request {

    private $Get;
    private $Post;
    private $Files;
    private $Server;
    private $Cookie;
    private $Request;

    public function __construct(){
        $this->Get = $_GET;
        $this->Post = $_POST;
        $this->Files = $_FILES;
        $this->Server = $_SERVER;
        $this->Cookie = $_COOKIE;
```

```
        $this->Request = $_REQUEST;
        unset($_SERVER, $_GET, $_POST, $_FILES, $_COOKIE, $_REQUEST);
    }

    public function getHost() {
        ...
    }

    public function getHostSSL() {
        ...
    }

    public function getHostAddress() {
        ...
    }

    public function getUri() {
        ...
    }

    public function getUriSegments() {
        ...
    }

    public function getMethod() {
        ...
    }

    public function getQueryString() {
        ...
    }

    public function getParams($type, $key = null){
        ...
    }

    public function decode($string){
        ...
    }
}
```

CONCLUSION

Construire un framework PHP personnalisé est une formidable occasion d'apprentissage. En structurant votre code sous forme de classes et de modules, vous comprenez mieux comment chaque partie d'un framework s'imbrique — qu'il s'agisse de la configuration, de la gestion des routes, de la journalisation ou de la gestion des requêtes. Cette approche modulaire contribue également à garder l'application légère et maintenable, en chargeant uniquement les modules nécessaires.

Même si ce projet est encore en cours, cette esquisse devrait vous aider à prendre un bon départ. Libre à vous d'ajouter de nouveaux modules (par exemple, un routeur, un gestionnaire de base de données ou un moteur de template) et d'affiner le code. En fin de compte, que vous choisissiez un framework connu ou que vous écriviez le vôtre, comprendre ces briques de base ne fera qu'aiguiser vos compétences.

J'espère que cela vous a donné des pistes ! Restez à l'affût de nouvelles mises à jour au fil du développement de ce mini-framework. En attendant, n'hésitez pas à partager vos idées ou à poser des questions dans les commentaires.

MOTS-CLÉSGENERALCORE-FRAMEWORK-FRFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN