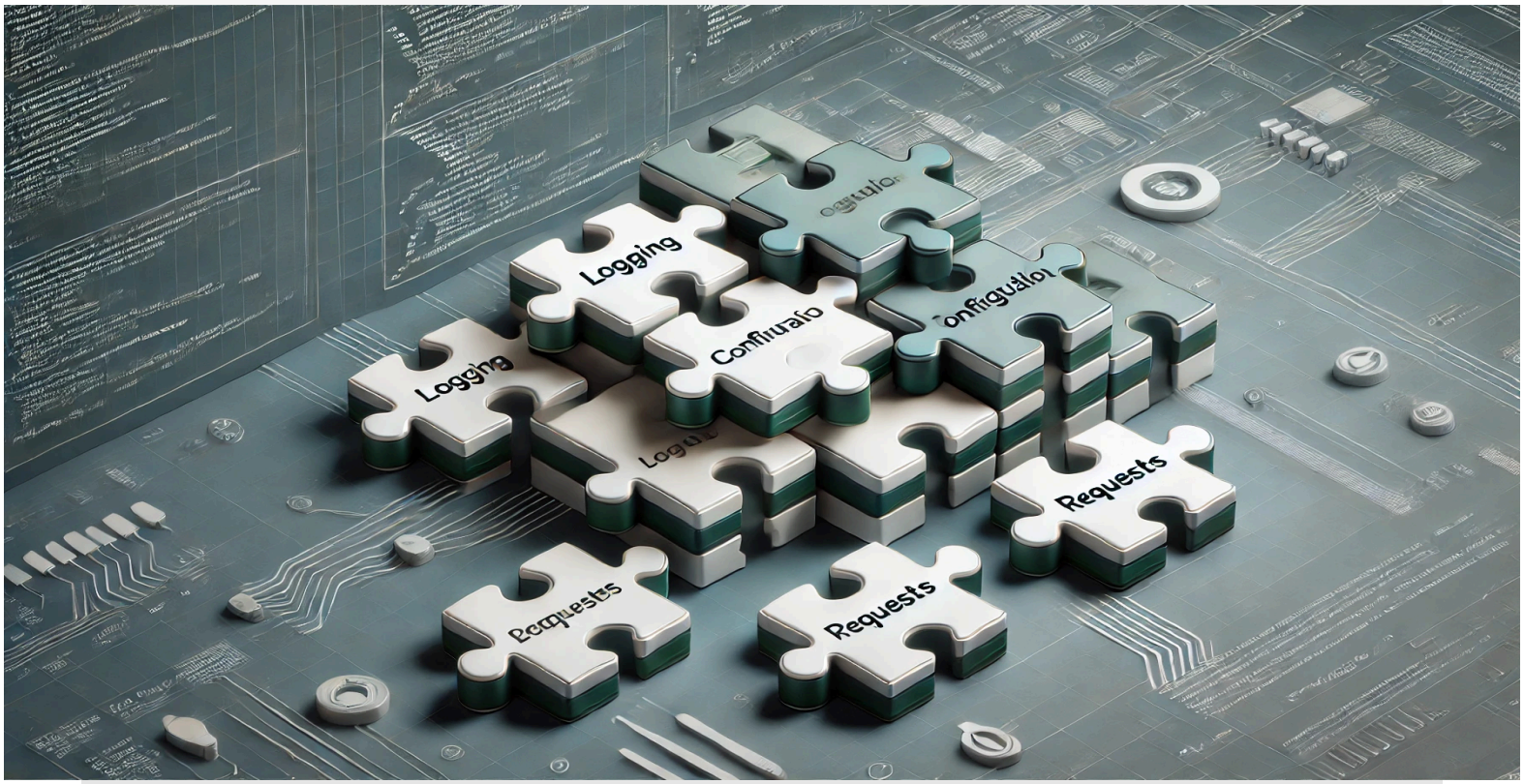
TABLE OF CONTENTS

WHY A MODULAR APPROACH? 4
DEFINING OUR CORE MODULE (“CORE”) 4
BOOTSTRAP CLASS 4
FALLBACK MODULE CLASS 7
CONFIGURATION CLASS (REPLACING “CONFIGURATOR”) 8
LOGGER CLASS (CORELOGGER) 10
REQUEST CLASS 12
CONCLUSION 14
RELATED ARTICLES 15
TAGSGENERALCORE-FRAMEWORK-ENFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN 15



LET'S TALK - BUILDING A MODULAR PHP FRAMEWORK FROM SCRATCH

Author(s): Louis Ouellet

Have you ever worked with popular PHP frameworks like CakePHP or Symfony and thought, “I wonder how these were built?” PHP frameworks can be tremendous time-savers, but they are truly powerful only when you know them inside and out. Developing your own mini-framework can be a great learning exercise, giving you deeper insight into best practices, modularity, and maintainability.

In this article, I share how I’ve started building my own PHP framework from the ground up. This includes constructing a Bootstrap class, handling configuration, setting up modules, creating a logging system, and finally wrapping requests into a tidy Request class. By walking through each piece, you’ll see the value of a well-structured, modular approach that can be extended with custom modules as needed.

WHY A MODULAR APPROACH?

Modularity is key for creating lightweight, flexible applications. A large framework can introduce overhead if you only need one or two components out of many. By splitting functionality into separate modules that can be selectively loaded, you ensure performance and maintainability.

DEFINING OUR CORE MODULE (“CORE”)

I’ve decided to name the main module simply `core`. This module includes any base classes that other modules may need, plus the core logic for initializing the application itself. An important detail: we want each module to initialize only once.

BOOTSTRAP CLASS

The `Bootstrap` class manages which modules are loaded, depending on the application’s scope (e.g., Router, CLI, API). Below is the work-in-progress. Notice that we use a default configuration array and allow for custom overrides via our configuration system (shown later):

Source: Bootstrap.php

src/Bootstrap.php

```php
<?php

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

```php
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();
            }
        }
    }
}
```

We then test the result with a short script:

Source: test.php

test.php

```
<?php

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

FALLBACK MODULE CLASS

One downside to having optional modules is that you need to check whether each one has been loaded (i.e., not null). Instead, we can create a default `Module` class that notifies the developer if a method is called on a non-installed module:

Source: Module.php

src/Module.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;
        }
}
```

This approach gracefully halts and logs an error if the developer tries to use an uninstalled module.

CONFIGURATION CLASS (REPLACING “CONFIGURATOR”)

Next, let’s look at our updated `Config` class (formerly `Configurator`). This class manages our JSON-based configuration files:

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;

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) {
```

```
        ...
    }
}
```

LOGGER CLASS (CORELOGGER)

Below is a simplified `Logger` class that uses our new `Config` class to manage settings (like log level, rotation, etc.). It demonstrates how to chain methods for more streamlined logging:

Source: Log.php

src/Log.php

```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(){
        ...
    }
}
```

REQUEST CLASS

Finally, we have a `Request` class that wraps the usual superglobals (`$_GET`, `$_POST`, etc.) into a more controlled interface. We even unset the original variables to ensure developers call the class methods instead. This fosters a centralized place for data sanitization or transformation.

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

Building a custom PHP framework is a fantastic learning opportunity. By organizing your code into classes and modules, you gain a deeper understanding of how each part of a framework works—configuration, routing, logging, and request handling. This modular approach also helps keep your application lightweight and maintainable, as you can load only the modules you need.

Although this is still a work in progress, the outline above should help you get started. From here, you can add new modules (for example, a router, database handler, or templating engine) and continue refining the code. Ultimately, whether you stick to popular frameworks or roll your own, knowing how these building blocks fit together can only sharpen your skills.

I hope this provided some insights! Stay tuned for more updates as I continue developing this mini-framework. In the meantime, feel free to share your ideas or ask any questions in the comments.

RELATED ARTICLES

- Let's Talk - Building a Modular PHP Framework from Scratch
- Let's Talk - Building a Modular PHP Framework part 2
- Let's Talk - Building a Modular PHP Framework part 3

TAGSGENERALCORE-FRAMEWORK-ENFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN

- Twitter
- Facebook
- LinkedIn
- Reddit
- Telegram
- Email

View the discussion thread.

From:
https://laswitchtech.com/ - **LaswitchTech**

Permanent link:
https://laswitchtech.com/en/blog/2025/01/24/let-s-talk-building-a-modular-php-framework-from-scratch

Last update: **2026/02/16 13:21**