Configuration in PHP applications

19 min read · Last change January 21, 2018

What is configuration?

Applications require a centralized place where settings are stored. All the values stored here are required to configure the behavior of the application and to define the resources for other entities of the application. These include usernames, passwords, database access info, API keys, email settings and similar.

The main purpose of configuration is the integrity of the stored values, the reachability of values for a certain domain, and the static behavior.

Configuration is the domain-aware orchestration of static settings, which shouldn't change between a request and a response.

What should a configuration implementation cover?

Configuration is a separate concern that combines multiple responsibilities into one implementation. All these responsibilities should have its own classes due to the Single responsibility principle.

Distribution (mandatory)

One of the configuration responsibilities is distribution. Distribution covers the reachability of the values in a configuration implementation. It ensures the availability management of configuration domains inside the configuration. To avoid any side effects: Distribution should be implemented immutable.

For example, database settings are grouped into a database domain to ensure that the database object always contains only database related settings.

Validation and sanitization (mandatory)

Another responsibility is validation and sanitization. Both cover the integrity of values in a configuration implementation.

For example, database settings have different types of settings. Validation ensures that the given values fit their required types. Sanitization ensures that resources are converted to their required types prior to validation.

Zero-Configuration (optional)

The responsibility of Zero-Configuration is an optional responsibility that automatically enforces default values to a configuration. The reason behind this approach is to reduce the effort of proper application configuration.

For example, database settings are traditionally limited to the hostname, port and the credentials that are required to authenticate to the database service.

The Zero-Configuration approach enforces the default values. In case of MySQL database:

```
hostname: localhost port: 3306
```

With these default values in mind, defining hostname or port is not needed in common cases, when the targeted service is located at localhost and listening on port 3306.

Caching (recommended)

The responsibility of caching is a recommended responsibility that should be always implemented in a common web application to ensure validity, integrity and availability in the fastest way.

Formats

Application configuration can be defined in all sorts of formats and places. From the regular PHP files, to other file formats such as YAML, INI, XML, JSON, NEON and similar. It can be defined even in the database.

The format of the defined application depends on the complexity of what must be configured and the wanted complexity of the configuration definition. Some configuration formats are limited to a specific set of definition utilities, other formats are open side effects to the user land that may have an impact to the configuration integrity.

Choose the configuration format based on these suggestions and what is suitable for your project case or better readability for you.

PHP files

```
<?php // config/config.php $configuration = [ 'database' => [ 'hostname' => 'localhost', 'port' => 3306 'name' => 'db_name',
'username' => 'db_username', 'password' => 'db_secret_password', ] ];
```

PHP files are actually scripts that define an object, array or a mixture of both.

The major benefit of this format is the validity of the file format which is directly enforced by the PHP parser. If opcode caching is available, this format will be also pretty fast.

The major downside of this format is the abuse possibility. You can quickly invalidate the static state of a configuration definition by utilizing conditions or other stuff that might change the returning value which is sensitive to the environment or request.

YAML

```
# config/config.yml database: hostname: 'localhost' port: 3306 name: 'db_name' username: 'db_username' password:
'db_secret_password'
```

YAML is a format that unifies the benefits of JSON and XML into a single but different format. YAML is both, a format that is intended to act as a data notation and as a file format to define documents. YAML has also a lot of different features that are not common to documents or object notations (entity linking). YAML supports widely the same syntax as JSON does.

To parse YAML files there are available 3rd party libraries such as Symfony Yaml Component, or the Yaml PHP Extension, which isn't bundled with PHP.

INI

```
; config/config.ini [database] database_hostname=localhost database_port=3306 database_name=db_name database_username=db_username
database_password=db_secret_password
```

INI files define data in a simple way. The INI format is less complex than other formats. Whatever you want to assign to a field would be associated as it would be associated in PHP. INI files can have groups as well, but does not support higher structs (arrays, objects).

Parsing of INI files can be done with PHP parse_ini_file() and pase_ini_string() functions.

XML

```
<?xml version="1.0" encoding="UTF-8"?> <configuration> <database default="true"> <server hostname="localhost" port="3306">
<environment dbname="db_name"> <auth username="db_username" password="db_secret_password" /> </environment> </server> </database>
</configuration>
```

XML documents relay on specific entities and have a Document Object Model that requires a lot of knowledge to define proper documents. XML files are prone to failure due to its high efforts to define the document.

The major benefit of XML documents is they can easily transported to other formats or structures using XSLT or traditional scripts. XML is the most flexible format because of the descriptive nature of XML.

The major downside of XML is, it is not intended to be human readable as other formats are.

There are multiple ways to parse XML format. For example, there is XML Parser extension enabled by default.

JSON

```
{ "database": { "hostname": "localhost", "port": "3306", "name": "db_name", "username": "db_username", "password":
"db_secret_password" } }
```

JSON is actually an Object Notation not a real file format. Since composer enters the PHP universe, JSON is the considered data structure for statically applied configuration in files with the file extension: `.json`.

The major benefit of JSON is, it remains always to the same syntax and does restrict the configuration definition context to strings, integers, floats and booleans.

The major downside of JSON is, it is an object notation. No inline documentation allowed and you actually configure your application with a format that was intended to be used as to transport data from end-point to end-point.

To parse JSON format, there is available json_decode().

NEON

```
# config/config.neon database: mysql( hostname=localhost, port=3306, name=db_name, username=db_username,
password=db_secret_password )
```

NEON is a YAML derivate invented by the guys of the Nette Framework, that adds entities to the definition model of YAML.

Package nette/neon can parse NEON files.

Performance

Configuration formats must be processed by PHP so various formats can have different performance. It may seem that the fastest way is to use PHP format since PHP understands it by default, however you can use different caching strategies when parsing configuration files to PHP understandable formats (PHP files with arrays or classes).

The performance of configurations depends on the implementation not on the format utilized in general. It does not matter if you use XML, YML, NEON or plain PHP when the application utilizes a cache for configuration where the distribution responsibility products are cached. You should always consider a caching mechanism when implementing configurations into your application.

There are 2 well known caching approaches:

Active caching

Active caching observes always configuration files each time your application is bootstrapped. Once a configuration file has changed, the configuration cache invalidates and a rebuild process for the cache will be executed.

Active caching raised performance peeks to your application whenever the configuration cache invalidates a segment (domain) of your configuration. Active caching is the caching mechanism to choose when performance peaks do not deal damage to the overall application performance (small applications with less traffic).

Passive caching

Passive caching relays on maintainer actions to enforce the cache building for the application after changes are made to the configuration files.

Passive caching ensures that no performance peeks hit your application. The built configuration cache is always committed to the server by the maintainer of the application. Passive caching is the caching mechanism to choose when aiming for a very balanced application performance (applications with high traffic).

Environments

Applications are used in different environments (development, production, staging, testing, beta, etc.). When developing an application, the configuration is different from the configuration defined on the production server. To approach this effectively, best practice is to have different configuration files for each environment and add/upload them to the project separately from the secure location or by using one of the deployment tools which can do that.

You can also define default values for all environments that get overwritten when deploying or installing application locally.

```
# config/config.yml.dist database: hostname: 'localhost' port: 3306 name: 'project' username: 'root' password: ~
```

Many projects use the practice of adding `dist` to the filename which means the default configuration that comes with the distribution. For example, the hostname defaults to `localhost`, port defaults to MySQL default port `3306` etc. The special character tilde - ~ above notes the `null` value in YAML. All these settings can be overridden and set in your application in `config/config.yml` when installing.

Types of configuration

Types of application configuration can be structured into the following types:

- **Infrastructure configuration**

  These depend on the environment where the application is running and don't define the behavior of the application directly. This includes security credentials, such as database passwords, API access keys and similar.

- **Application configuration**

  These define the behavior of the application and depend on the environment where the application is running. For example, the debugging turned on or off, database type (for example, you can use different database type in testing), locale settings and similar.

  - Fixed application configuration

    These don't change very often during the certain application version. For example database type, number of items shown per page etc.

  - Variable application configuration

    These change more frequently in certain application version. For example user settings (showing/hiding signature in forum topics, default currency used in e-store for signed in users), settings meant to be changed by non-developers (contact emails, Google sitemap settings) etc.

Bad practices

Using PHP constants to define configuration values might seem like a good choice because of the global state:

```
<?php // config/config.php define('DATABASE_NAME', 'db_name'); define('DATABASE_USERNAME', 'db_username');
define('DATABASE_PASSWORD', 'db_password');
```

However this is not a good practice for the following reasons:

- You are polluting the global namespace and can have compatibility issues from other libraries or code that might define same constant.
- Their value cannot be changed in the code, which might have issues when testing code.
- Difficult code refactoring in case they were used in multiple places
- Limited set of types available (only boolean, integer, float, string, array and resource).

Some of the limitations mentioned above can be avoided by using class constants. This should be used only for configuration values that never or very rarely change in certain application version. For example, maximum number of elements shown per page and similar:

```
<?php class Article { const MAX_ITEMS_PER_PAGE = 10; //... } $limit = Article::MAX_ITEMS_PER_PAGE;
```

Limitation of this is still difficult changing of these values in testing environment for example.

Singleton pattern

Another approach for storing configuration values is to use singleton pattern because it introduces global state and simple access to the configuration values in the application:

```
<?php class Config { /** * @var Config */ private static $instance; /** * @var array */ private static $values = []; /** *
Instantiation can be done only inside the class itself */ private function __construct() {} /** * @return Config */ public static
function getInstance() { if (!isset(self::$instance)) { self::$instance = new self(); } return self::$instance; } /** * Set
configuration value by key. * * @param string $key * @param mixed $value */ public static function set($key, $value) {
```

```
self::$values[$key] = $value; } /** * Get configuration value by key. * * @param string $key * @return mixed */ public static function get($key) { if (isset(self::$values[$key])) { return self::$values[$key]; } return null; } /** * Cloning singleton is not possible. * * @throws Exception */ public function __clone() { throw new Exception('You cannot clone singleton object'); } }
$config = Config::getInstance(); $config->set('database_username', 'db_username');
```

However using singleton pattern reduces testability as well. Instead, a better practice is to use the dependency injection, container and repository patterns.

Misconception of configurations

Everything that is not static in the moment the configuration is defined should not be considered as configuration. You should always utilize the format that allows you to define exempts (like entities in NEON) that provides a structure that allows the implementation to fit the setting to an environment's sensitive state.

A neon example:

```
# general debug: whenIn(server=REMOTE_ADDR, [::1, 127.0.0.1], if=true, else=false) environment: sapi({ php_cli: cli }, default=www);
```

Translation data should not be part of the configuration, no matter which kind of translation distribution format was chosen. Translations always relay on a non static part.

Security

When working with application configuration, never expose sensitive configuration files in public. To avoid that, place the configuration files outside the publicly accessible document root on the server, so they are not accessible over web `https://example.com/config/config.yml`.

Folder structure could be in this case the following:

```
project/ public/ index.php css/ js/ images/ config/ config.yml src/ vendor/
```

The `public` folder in this case is the document root folder which is accessible over web `https://example.com`.

Environment variables

A good practice is to use environment variables for configuration such as security credentials (database access, API keys, etc.). Environment variables are special system variables defined on the system level.

First a bit of an introduction into PHP environment variables and some caveats when working with them.

On Apache servers environment variables can be defined in the `VirtualHost` configuration with the special `SetEnv` directive of mod_env:

```
<VirtualHost *:80> ServerName example.com DocumentRoot "/var/www/project/public" DirectoryIndex index.php index.html SetEnv APP_DATABASE_USERNAME db_username SetEnv APP_DATABASE_PASSWORD db_secret_password <Directory "/var/www/project/public"> AllowOverride All Allow from All </Directory> </VirtualHost>
```

On Nginx servers environment variables can be set with `fastcgi_param` directive in configuration file where the `fastcgi_params` is being included:

```
# /etc/nginx/sites-available/example.com location ~ \.php$ { #... include fastcgi_params; fastcgi_param APP_DATABASE_USERNAME db_username; fastcgi_param APP_DATABASE_PASSWORD db_secret_password; #... }
```

When your application is running in PHP CLI (which does not use web server), environment variables must be set also with `export` (for Linux servers):

```
export APP_DATABASE_USERNAME="db_username" export APP_DATABASE_PASSWORD="db_secret_password"
```

In PHP environment variables can be than accessed with getenv():

```
<?php // config/config.php $configuration = [ 'database_username' => getenv('APP_DATABASE_USERNAME'), 'database_password' => getenv('APP_DATABASE_PASSWORD'), ];
```

PHP dotenv

A very useful PHP library that adds best practices to your application when working with environment variables for configuration is PHP dotenv.

After installation with Composer:

```
composer require vlucas/phpdotenv
```

Add the `.env` file to the root of your project:

```
APP_DATABASE_USERNAME="db_username" APP_DATABASE_PASSWORD="db_secret_password"
```

In PHP the configuration values can be than accessed like environment variables explained above:

```
<?php require __DIR__.'/../vendor/autoload.php'; $dotenv = new Dotenv\Dotenv(__DIR__.'/..'); $dotenv->load(); $dbUsername = getenv('APP_DATABASE_USERNAME');
```

Worth noting is that environment variables are still exposed on the system level. Be careful to not output them. For example, with `phpinfo()`. Important to understand is, when the environment variables are useful for your case scenario and when to use other tools like Vault, Chef or similar.

Git repositories

When committing code to the source control (Git), avoid adding configuration files to the commits. In case of Git, ignore the configuration files containing sensitive configuration values with `.gitignore`:

```
#.gitignore file which omits committing config.php file to the Git repository /config/config.php
```

Encapsulation

The infrastructure configuration don't change during the running of the application. In case you don't have simple access to the production environment infrastructure configuration values (database credentials) but still need to develop application independently and frequently add more infrastructure related configuration in the next version of the application, you can use encapsulation:

```
<?php // config/database.php return [ 'database_name' => 'db_name', 'database_username' => 'db_username', 'database_password' => 'db_secret_password', ];
```

And use it in your application configuration with the rest of the infrastructure configuration:

```
<?php // config/config.php return array_merge([ 'fb_app_id' => 123456, 'fb_app_secret' => 'facebook_app_secret', ], require(__DIR__.'/database.php') );
```

Configuration stored in the database

Some configuration values that often change or are meant to be changed by non-developers, can be defined in the database so they can be easily changed over the UI.

There are multiple different approaches you can look into. From using key-value storages to designing the database schema for these tables accordingly for the current project:

- Key-value table (column types can be json, array or similar for different configuration types)
- EAV (entity-attribute-value)
- Configuration in the same table as the other entities (for example, user settings)
- ... and many other ideas

Serialization is useful for storing values without losing their type and structure. Above PHP configuration example can be represented in serialized format using the serialize():

```
a:3:
{s:13:"database_name";s:7:"db_name";s:17:"database_username";s:11:"db_username";s:17:"database_password";s:18:"db_secret_password";}
```

SQLite

SQLite is file-based fast. It is also valuable as a configuration resource and therefore valuable to store configurations.

The major benefit of SQLite is, its table can contract value types, which exclude the need of validating the types of configuration values. Those value types are enforced inbound and output to the database.

The major downside of SQLite is, it is a binary format and not maintainable without an SQLite client.

How to use configuration in PHP application?

Many times you might be tempted to access the configuration values in the application code directly:

```
<?php class Database { private $name, $username, $password; /** * Constructor. */ public function __construct() { $this->name = getenv('APP_DATABASE_NAME'); $this->username = getenv('APP_DATABASE_USERNAME'); $this->password = getenv('APP_DATABASE_PASSWORD'); } //... }
```

Issue with such approach is difficult maintaining of the code when scaling and testing. The other not good enough step is injecting the configuration handler directly in the needed class:

```
<?php class Config { /** * @var array */ private $values; /** * Constructor. * * @param array $values */ public function __construct($values) { $this->values = $values; } /** * Get configuration value by key. * * @param string $key * @return mixed */ public function get($key) { return $this->values[$key]; } } class Database { private $name, $username, $password; /** * Constructor. * * @param Config $config */ public function __construct($config) { $this->name = $config->get('database_name'); $this->username = $config->get('database_username'); $this->password = $config->get('database_password'); } } // ... $config = new Config(require(__DIR__.'/../config/config.php')); $database = new Database($config);
```

Instead, you should inject the needed configurations separately to the separate adapter class:

```
<?php class DatabaseAdapater { protected $inhibitor = null; protected $instance = null; private $name; private $username; private $password; private $hostname = '127.0.0.1'; /** * Constructor. */ public function __construct() { $this->inhibitor = Closure::bind( function ($name = null, $username = null, $password = null, $hostname = null): PDO { return new PDO( 'mysql:dbname='.($name ?? $this->name).';host='.($hostname ?? $this->hostname), $username ?? $this->username, $password ?? $this->password ); }, $this, DatabaseAdapter::class ); } /** * Set database name. * * @param string $name */ public function setName(string $name) { $this->name = $name; } /** * Set database username. * * @param string $username */ public function setUsername(string $username) { $this->username = $username; } /** * Set database password. * * @param string $password */ public function setPassword(string $password) { $this->password = $password; } /** * Set database hostname. * * @param string $hostname */ public function setHostname(string $hostname) { $this->hostname = $hostname; } /** * Get Database adapter instance. * * @return DatabaseAdapter */ public function getInstance(): PDO { if ($this->instance instanceof PDO) { return $this->instance; } return $this->instance = call_user_func($this->inhibitor); } } class Database { /** * @var DatabaseAdapter */ private $adapter; /** * Constructor. * * @param DatabaseAdapter $adapter */ public function __construct(DatabaseAdapter $adapter) { $this->adapter = $adapter; } } $adapter = new DatabaseAdapter(); $adapter->setName($config->get('database_name')); $adapter->setUsername($config->get('database_username')); $adapter->setPassword($config->get('database_password')); $db = $adapter->getInstance();
```

Conclusion

Configuration should always be validated to ensure its integrity. The performance of configuration is not based on the size or the format in which the configuration has been defined in. It is based on the mechanism that aggregates the configuration distribution to the application. Always cache a process that might result in performance peaks, choose the right caching mechanism for your application and configuration size.

See also

More resources you should look into:

- Nette Bootstrap - Configuration component from Nette Framework.
- PHP dotenv - PHP library which loads environment variables.
- Respect/Config - A tiny, fully featured dependency injection container as a DSL.
- Symfony Config component
- Twelve Factor App - Configuration chapter of the Twelve Factor App book.
- werx/config - Use environment-specific configuration files in your app.
- Zend Config

Security

How to secure PHP web applications and prevent attacks? How to work with users' passwords and how to securely hash passwords in PHP? What is SQL injection and how to prevent it? How to securely upload files with PHP? Configuration in PHP applications How to protect and hide PHP source code? How to install an SSL certificate and enable HTTPS? Encryption, hashing, encoding and obfuscation

Found a typo? Something wrong with this content?

Just fork and edit it.

About PHP.earth

Sitemap Team Get Involved Status

PHP.earth documentation

Index PHP installation wizard FAQ <?php tips

Community

Facebook Group GitHub Slack Twitter