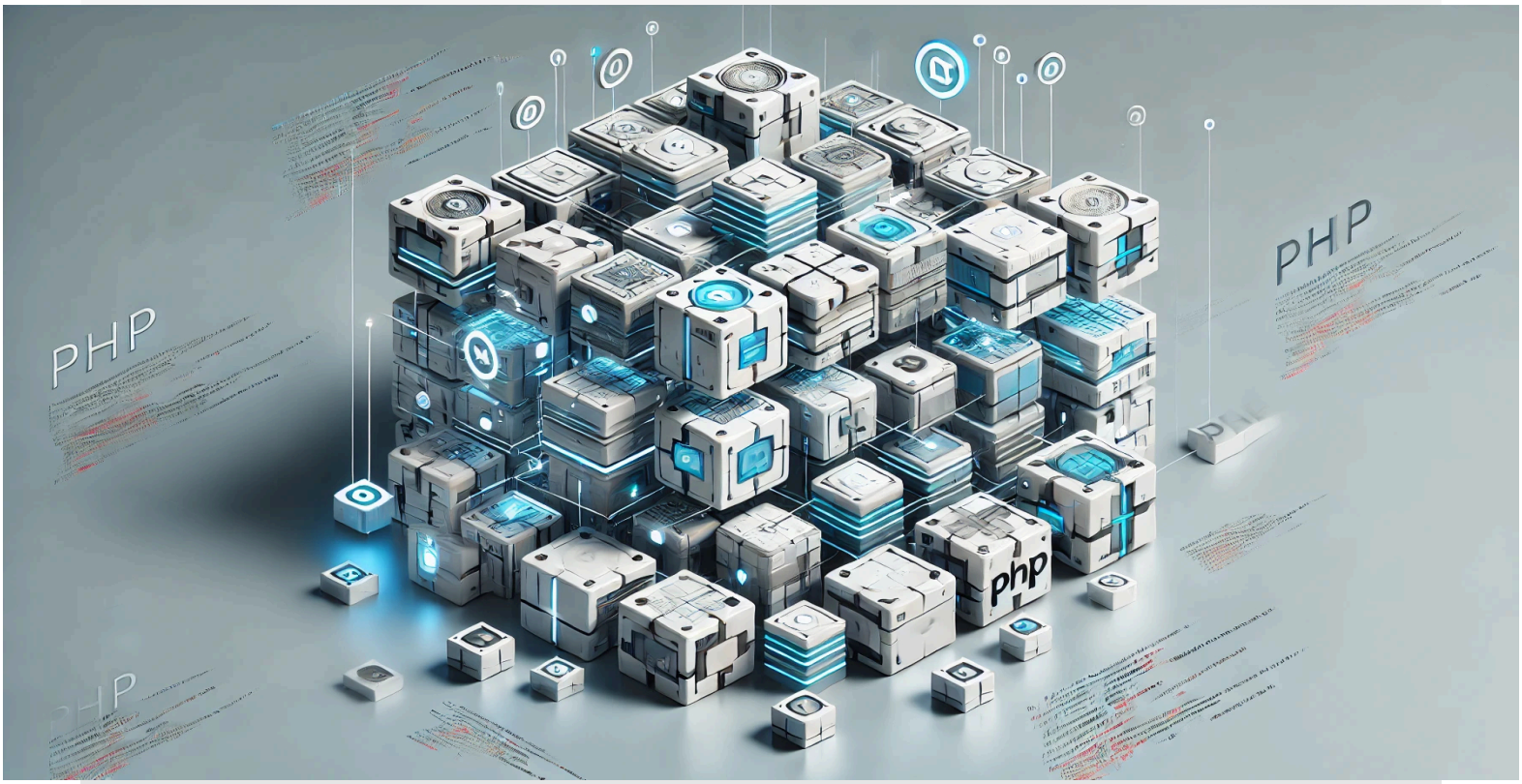
TABLE DES MATIÈRES

CROSS-SITE REQUEST FORGERY (CSRF) 4
CONFIGURATION D'APACHE (.HTACCESS) 8
CONSTRUCTION DU MODULE API 9
GESTION DE LA BASE DE DONNÉES 15
LA CLASSE DATABASE 15
LES CLASSES CONNECTOR ET MYSQL 18
LA CLASSE QUERY 26
LES CLASSES SCHEMA ET DEFINITION 40
CONCLUSION 60
DÉPÔT GITHUB 61
TAGSGENERALCORE-FRAMEWORK-FRFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN 61



PARLONS-EN - CONSTRUIRE UN FRAMEWORK PHP MODULAIRE, PARTIE 3

Auteur(s) : Louis Ouellet

Dans ce troisième volet de notre série **Construire un Framework PHP Modulaire**, nous allons aborder :

- La protection contre les attaques **Cross-Site Request Forgery (CSRF)**
- La création d'une **API** et de ses **Endpoints**
- La gestion de la base de données, incluant :
  - La gestion de la structure de la base via **Schema**
  - La gestion des requêtes via **Query**

Nous allons poursuivre le travail entamé précédemment, en mettant l'accent sur la sécurité (CSRF), la configuration des routes pour l'API et une gestion plus avancée de la base de données.

CROSS-SITE REQUEST FORGERY (CSRF)

Qu'est-ce que le CSRF ? Le **Cross-Site Request Forgery (CSRF)** est une vulnérabilité de sécurité Web qui permet à un attaquant de duper un utilisateur afin qu'il effectue des actions non désirées sur un site ou une application Web où il est déjà authentifié. En d'autres termes, l'attaquant exploite la session du navigateur de l'utilisateur (y compris les cookies) pour envoyer des requêtes non autorisées, souvent sans que l'utilisateur s'en rende compte.

Pour se protéger contre le CSRF, nous générons et validons un jeton secret qui doit être inclus dans toutes les requêtes qui ne sont pas de type GET.

Nous nous assurons d'abord que la session est démarrée dans la classe **Bootstrap** :

```php
// Start the session
if (!defined('STDIN') && session_status() === PHP_SESSION_NONE) {
    ini_set('session.cookie_samesite', 'Strict');
    ini_set('session.cookie_secure', 'On');
    session_start();
}
```

Ensuite, nous créons la classe **CSRF** :

Source: CSRF.php

src/CSRF.php

```php
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;
```

```php
// Import additionnal class into the global namespace
use Exception;

class CSRF {

    const FIELD = 'csrf';
    const LENGTH = 32;
    const ROTATION = true;

    // Global Properties
    protected $Request;
    protected $Output;
    protected $Config;

    // Properties
    protected $Token = null;
    protected $Field = self::FIELD;
    protected $Length = self::LENGTH;
    protected $Rotate = self::ROTATION;

    /**
     * Create a new CSRF instance.
     *
     * @param  string|null  $field
     * @return void
     */
    public function __construct(){

        // Import Global Variables
        global $REQUEST, $OUTPUT, $CONFIG;

        // Initialize Properties
        $this->Request = $REQUEST;
        $this->Output = $OUTPUT;
        $this->Config = $CONFIG;

        // Add the csrf config file
        $this->Config->add('csrf');

        // Retrieve and Set CSRF Settings
        $this->Field = $this->Config->get('csrf', 'field') ??
$this->Field;
```

```
        $this->Length = $this->Config->get('csrf', 'length') ??
$this->Length;
        $this->Rotate = $this->Config->get('csrf', 'rotate') ??
$this->Rotate;

        // Check if the method used should be validated
        if(!defined('STDIN') && in_array($this->Request->getMethod(),
['POST', 'PUT', 'PATCH', 'DELETE'])){
            if(!$this->validate($this->Request->getParams('POST',
$this->Field) ?? null)){
                $this->Output->print(
                    'Invalid CSRF Token',
                    array('Content-Type: application/json', 'HTTP/1.1
403 Forbidden'),
                );
            }
        }
    }

    /**
     * Generate token.
     * @return $this
     */
    protected function generate(){

        // Retrieve the existing Token
        $token = $this->Request->getParams('SESSION', $this->Field) ??
bin2hex(random_bytes((int) ($this->Length / 2)));

        // Store the token
        $this->Token = $token;
        $this->Request->setParams('SESSION', $this->Field,
$this->Token);

        // Return
        return $this;
    }

    /**
     * Clear token.
     * @return $this
     */
```

```
    protected function clear(){

        // Check if rotate is enabled
        if($this->Rotate){

            // Clear the token
            $this->Token = null;
            $this->Request->setParams('SESSION', $this->Field, null);
        }

        // Return
        return $this;
    }

    /**
     * Get token.
     * @return string $this->Token
     */
    public function token(){
        return $this->generate()->Token;
    }

    /**
     * Validate a token.
     * @param  string  $token
     * @return boolean
     */
    public function validate(?string $token){
        $this->generate();
        if(!empty($token) && hash_equals($token, $this->Token)){
            $this->clear()->generate();
            return true;
        }
        return false;
    }

    /**
     * Generate a hidden input field.
     */
    public function field(){
        return '<input type="hidden" name="' . $this->Field . '"
value="' . $this->token() . '">';
    }
```

```
}
```

La classe CSRF :

- Récupère la configuration de l'application ou utilise des valeurs par défaut.
- Valide le jeton pour toutes les requêtes autres que GET.
- Génère un nouveau jeton s'il n'en existe pas, et le stocke en session.
- Fournit une méthode `field()` pour injecter un champ caché CSRF dans vos formulaires.

CONFIGURATION D'APACHE (.HTACCESS)

Pour que notre application puisse gérer correctement les routes, nous créons deux fichiers **.htaccess** : l'un à la racine (**root**) et l'autre dans le sous-répertoire **webroot**.

Source: .htaccess

.htaccess

```
AddType application/javascript .mjs

<IfModule mod_headers.c>
  RequestHeader unset Proxy
</IfModule>

<IfModule mod_rewrite.c>
  RewriteEngine on
  RewriteRule ^(\.well-known/.*)$ @@ [L]
  RewriteRule ^$ webroot/ [L]
  RewriteRule (.*) webroot/@@ [L]
</IfModule>
```

Source: .htaccess

webroot/.htaccess

```
AddType application/javascript .mjs

<IfModule mod_headers.c>
  RequestHeader unset Proxy
</IfModule>

<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^(.+)$ index.php [QSA,L]
  RewriteRule ^cli - [F,L]
  RewriteRule ^.htaccess - [F,L]
</IfModule>
```

Ces règles veillent à ce que les appels soient correctement dirigés vers **index.php**, et bloquent l'accès direct à certains fichiers (CLI, .htaccess).

CONSTRUCTION DU MODULE API

Le module **API** nous permet (et éventuellement à des tiers) de demander, d'échanger ou de synchroniser des données avec notre application. Notre logique se trouve principalement dans deux fichiers : **API.php** et **Endpoint.php**.

Source: API.php

src/API.php

```
<?php

/**
 * Core Framework - API
 *
```

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use Exception;

class API {

    // Global Properties
    protected $Output;
    protected $Request;
    protected $Config;
    protected $Auth;

    /**
     * Constructor
     */
    public function __construct() {

        // Import Global Variables
        global $OUTPUT, $REQUEST, $CONFIG, $AUTH;

        // Initialize Properties
        $this->Output = $OUTPUT;
        $this->Request = $REQUEST;
        $this->Config = $CONFIG;
        $this->Auth = $AUTH;
    }

    /**
     * Execute the API Request
     */
    public function request(){

        // Retrieve the namespace
        $namespace = $this->Request->getNamespace();

        // Parse the namespace
```

```
        $parts = explode("/", trim($namespace, "/"));

        // Check if the namespace is valid
        if(count($parts) > 1){

            // Set the endpoint and method
            $endpoint = $parts[0];
            $class = ucfirst($endpoint) . "Endpoint";
            $method = $parts[1] . 'Action';

            // Set the endpoint path
            $path = $this->Config->root() . "/Endpoint/" .
ucfirst($endpoint) . "Endpoint.php";

            // Check if the endpoint exists
            if(!is_file($path)){

                // Set the path to the plugin path
                $path = $this->Config->root() . "/lib/plugins/" .
$endpoint . "/Endpoint.php";
            }

            // Check if the endpoint exists
            if(is_file($path)){

                // Load the endpoint
                require_once $path;

                // Check if the class exists
                if(class_exists($class)){

                    // Initialize the endpoint
                    $object = new $class();

                    // Check if the method exists
                    if(method_exists($object, $method)){

                        // Retrieve Auth Properties
                        $public = $object->getPublic();
                        $level = $object->getLevel();
                        $permission = "Endpoint>" . $namespace;

                        // Check if Auth is available and if the
```

```
endpoint is public
                        if(get_class($this->Auth) !== "Strap" &&
!$public){

                            // Check if the user is authenticated
                            if(!$this->Auth->isAuthenticated()){

                                // Send unauthorized
                                $this->Output->print('Unauthorized',
array('HTTP/1.1 401 Unauthorized'));
                            }

                            // Check if the user has the required
permission
                            if(!$this->Auth->hasPermission($permission,
$level)){

                                // Send forbidden
                                $this->Output->print('Forbidden',
array('HTTP/1.1 403 Forbidden'));
                            }
                        }

                        // Call the method
                        $results = $object->{$method}();

                        // Send the output
                        $this->Output->print($results['data'] ?? [],
array('HTTP/1.1 ' . $results['status'] ?? 500 . ' '. $results['message']
?? 'Internal Server Error'));
                    } else {

                        // Could not find the method, send not
implemented
                        $this->Output->print('Could not find the
method', array('HTTP/1.1 501 Not Implemented'));
                    }
                } else {

                    // Could not find the class, send not implemented
                    $this->Output->print('Could not find the endpoint',
array('HTTP/1.1 501 Not Implemented'));
```

```
                }
            } else {

                // Could not find the endpoint
                $this->Output->print('Could not find the endpoint',
array('HTTP/1.1 404 Not Found'));
            }
        } else {

            // Could not identify the Controller and/or Method, send bad
request
            $this->Output->print('Could not identify the Controller
and/or Action', array('HTTP/1.1 400 Bad Request'));
        }
    }
}
```

Source: Endpoint.php

src/Endpoint.php

```
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use Exception;

abstract class Endpoint {

    // Global Properties
```

```
    protected $Auth;
    protected $Model;
    protected $Helper;
    protected $Output;
    protected $Request;

    // Auth Properties
    protected $Level;
    protected $Public;

    /**
     * Constructor
     */
    public function __construct(){

        // Import Global Variables
        global $AUTH, $MODAL, $HELPER, $OUTPUT, $REQUEST;

        // Initialize Properties
        $this->Auth = $AUTH;
        $this->Model = $MODAL;
        $this->Helper = $HELPER;
        $this->Output = $OUTPUT;
        $this->Request = $REQUEST;
    }

    /**
     * Magic Method to catch all undefined methods
     * @param string $name
     * @param array $arguments
     * @return void
     */
    public function __call($name, $arguments) {

        // Send the output
        $this->Output->print('Endpoint not Implemented', array('HTTP/1.1
501 Not Implemented'));
    }

    /**
     * Get the public status
     */
```

```
    public function getPublic() {
        return $this->Public;
    }

    /**
     * Get the level
     */
    public function getLevel() {
        return $this->Level;
    }
}
```

Comment ça marche :

1. Nous analysons le namespace (chemin) de la requête pour déterminer l'endpoint et l'action.
2. Nous tentons de charger une classe correspondante dans le dossier `/Endpoint` ou dans un plugin.
3. Si elle existe, nous l'instancions et vérifions que la méthode demandée est disponible.
4. Des vérifications optionnelles d'authentification et de permissions sont effectuées.
5. L'action est exécutée et nous renvoyons le résultat en JSON avec le code HTTP approprié.

GESTION DE LA BASE DE DONNÉES

Dans la partie 2, nous avons introduit les **models**. Désormais, nous allons affiner la gestion de la base, incluant la création de requêtes et la gestion du schéma. Examinons brièvement les différentes classes.

LA CLASSE DATABASE

Sources: Database.php

src/Database.php

```php
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use LaswitchTech\Core\Connector\MySQL;
use LaswitchTech\Core\Connector\PostgreSQL;
use LaswitchTech\Core\Connector\SQLite;
use Exception;

class Database {

    /**
     * @var Config
     */
    private $Config;

    /**
     * @var Connector
     */
    private $connector;

    /**
     * Constructor
     */
    public function __construct()
    {

        // Import Global Variables
        global $CONFIG;

        // Initialize Properties
        $this->Config = $CONFIG;
```

```
        // Load the Database Configuration
        $this->Config->add('database');

        // Instantiate the appropriate connector
        switch($this->Config->get('database', 'connector')) {
            case 'mysql':
                $this->connector = new MySQL();
                break;
            default:
                throw new Exception('Invalid database connector');
        }

        // Connect to the database
        $this->connector->connect();
    }

    /**
     * Destructor
     */
    public function __destruct()
    {
        $this->close();
    }

    /**
     * Check if the database is connected
     *
     * @return bool
     */
    public function isConnected(): bool
    {
        if ($this->connector === null) {
            return false;
        }
        return $this->connector->isConnected();
    }

    /**
     * Close the connection
     */
    public function close(): void
    {
        if ($this->connector !== null) {
```

```
            $this->connector->close();
        }
    }

    /**
     * Create a new Query object
     *
     * @return Query
     */
    public function query(): Query
    {
        return new Query($this->connector);
    }

    /**
     * Create a new Schema object
     *
     * @return Schema
     */
    public function schema(): Schema
    {
        return new Schema($this->connector);
    }
}
```

Cette classe charge la configuration de la base de données et instancie le **Connector** adéquat en fonction du fichier de config. Elle fournit aussi des méthodes utilitaires pour créer de nouveaux objets **Query** ou **Schema**.

LES CLASSES CONNECTOR ET MYSQL

Sources: Connector.php

src/Connector.php

```
<?php
```

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use Exception;

abstract class Connector {

    /**
     * @var mixed
     */
    protected $Config;

    /**
     * Constructor
     *
     * Here you might load global config or do other setup tasks.
     * For this example, we assume there's a $CONFIG global object
     * that stores DB credentials, etc.
     */
    public function __construct()
    {
        global $CONFIG;
        $this->Config = $CONFIG;

        // If you have a method $CONFIG->add('database') to load DB
config, do that here.
        if (method_exists($this->Config, 'add')) {
            $this->Config->add('database');
        }
    }

    /**
     * Connect method to be overridden by child classes
     */
```

```
    public function connect()
    {
        // Implementation in child classes
    }

    /**
     * Close method to be overridden by child classes
     */
    public function close()
    {
        // Implementation in child classes
    }

    /**
     * Check if connected
     *
     * @return bool
     */
    public function isConnected(): bool
    {
        // Implementation in child classes
        return false;
    }

    /**
     * Execute a query
     *
     * @param string $sql
     * @return mixed
     */
    public function query(string $sql)
    {
        // Implementation in child classes
    }

    /**
     * Describe a table
     *
     * @param string $table
     * @return mixed
     */
    public function describe(string $table)
```

```
    {
        // Implementation in child classes
    }

    /**
     * Get the ID of the last inserted row
     *
     * @return int
     */
    public function lastId(): int
    {
        // Implementation in child classes
        return 0;
    }

    /**
     * Get the number of affected rows
     *
     * @return int
     */
    public function affectedRows(): int
    {
        // Implementation in child classes
        return 0;
    }

    /**
     * Prepare a query
     *
     * @param string $sql
     */
    public function prepare(string $sql, array $params = [])
    {
        // Implementation in child classes
        return false;
    }
}
```

Sources: MySQL.php

src/Connector/MySQL.php

```php
<?php

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

// Declaring namespace
namespace LaswitchTech\Core\Connector;

// Import additionnal class into the global namespace
use LaswitchTech\Core\Connector;
use Exception;
use mysqli;

class MySQL extends Connector {

    /**
     * @var mysqli
     */
    private $mysqli;

    /**
     * @var array
     */
    private static $describeCache = [];

    /**
     * Establish the MySQL connection
     *
     * @throws Exception
     */
    public function connect()
    {
        // In a real-world scenario, retrieve these from $this->Config
        // e.g. $dbConfig = $this->Config->get('database');
        // For demonstration, we'll just hardcode or assume values are
set in $dbConfig.
```

```php
        $dbConfig = $this->Config->get('database') ?? [];

        $host = $dbConfig['host'] ?? 'localhost';
        $user = $dbConfig['username'] ?? 'root';
        $pass = $dbConfig['password'] ?? '';
        $name = $dbConfig['database'] ?? 'demo';

        $this->mysqli = new mysqli($host, $user, $pass, $name);

        if ($this->mysqli->connect_error) {
            throw new Exception('MySQL Connection Error: ' .
$this->mysqli->connect_error);
        }
    }

    /**
     * Close the connection
     */
    public function close()
    {
        if ($this->mysqli) {
            $this->mysqli->close();
        }
    }

    /**
     * Check if connected
     *
     * @return bool
     */
    public function isConnected(): bool
    {
        return ($this->mysqli && !$this->mysqli->connect_errno);
    }

    /**
     * Execute a query
     *
     * @param string $sql
     * @return mixed
     * @throws Exception
     */
    public function query(string $sql)
```

```
    {
        $result = $this->mysqli->query($sql);

        if ($result === false) {
            throw new Exception('MySQL Query Error: ' .
$this->mysqli->error);
        }
        return $result;
    }

    /**
     * Describe a table
     *
     * @param string $table
     * @return mixed
     */
    public function describe(string $table)
    {
        if (!isset(self::$describeCache[$table])) {
            $result = $this->mysqli->query("DESCRIBE `$table`");
            self::$describeCache[$table] =
$result->fetch_all(MYSQLI_ASSOC);
        }
        return self::$describeCache[$table];
    }

    /**
     * Return the number of affected rows in last operation
     *
     * @return int
     */
    public function affectedRows(): int
    {
        return (int) $this->mysqli->affected_rows;
    }

    /**
     * Return the ID of the last inserted row
     *
     * @return int
     */
    public function lastId(): int
```

```
    {
        return (int) $this->mysqli->insert_id;
    }

    /**
     * Prepare a query
     *
     * @param string $sql
     */
    public function prepare(string $sql, array $params = [])
    {
        // 1) Prepare the statement
        $stmt = $this->mysqli->prepare($sql);
        if ($stmt === false) {
            throw new Exception('MySQL Prepare Error: ' .
$this->mysqli->error);
        }

        // 2) Build the type string.
        if (!empty($params)) {
            $types = '';
            foreach($params as $param) {
                if (is_int($param)) {
                    $types .= 'i';              // Integer
                } elseif (is_float($param)) {
                    $types .= 'd';              // Double
                } elseif (is_string($param)) {
                    $types .= 's';              // String
                } else {
                    $types .= 'b';              // Blob and Unknown
                }
            }

            // 3) Bind the parameters
            $stmt->bind_param($types, ...$params);
        }

        // 4) Execute
        if (!$stmt->execute()) {
            throw new Exception('MySQL Execute Error: ' . $stmt->error);
        }

        return $stmt;
```

```
    }
}
```

La classe **Connector** est abstraite : chaque type de base de données (par exemple MySQL, SQLite ou PostgreSQL) doit implémenter les méthodes requises. Dans notre exemple, seule la classe **MySQL** est développée.

LA CLASSE QUERY

Sources: Query.php

src/Query.php

```php
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use LaswitchTech\Core\Connector;
use Exception;

class Query {

    /**
     * List of valid SQL operators
     *
     * @var array
     */
```

```
    const operators = ['=', '!=', '<>', '>', '<', '>=', '<=', 'LIKE',
'NOT LIKE', 'IS NULL', 'IS NOT NULL'];

    /**
     * List of valid SQL conjunctions
     *
     * @var array
     */
    const conjunctions = ['AND', 'OR'];

    /**
     * List of valid SQL directions
     *
     * @var array
     */
    const directions = ['ASC', 'DESC'];

    /**
     * List of valid SQL join types
     *
     * @var array
     */
    const types = ['LEFT', 'RIGHT', 'INNER', 'FULL', 'SELF'];

    /**
     * @var Connector
     */
    private $connector;

    /**
     * Query type: 'select', 'insert', 'update', 'delete'
     *
     * @var string
     */
    private $type;

    /**
     * Fields to select (for SELECT queries)
     *
     * @var string
     */
    private $fields = '*';
```

```php
    /**
     * Table name
     *
     * @var string
     */
    private $table;

    /**
     * Index column
     *
     * @var string
     */
    private $index;

    /**
     * Limit of rows to return
     *
     * @var int
     */
    private $limit;

    /**
     * Filter key
     *
     * @var int
     */
    private $filter;

    /**
     * Array of WHERE clauses
     *
     * @var array
     */
    private $where = [];

    /**
     * Array of JOIN clauses
     *
     * @var array
     */
    private $join = [];
```

```php
    /**
     * Array of ORDER BY clauses
     *
     * @var array
     */
    private $order = [];

    /**
     * Key/value pairs for INSERT/UPDATE
     *
     * @var array
     */
    private $values = [];

    /**
     * Parameters for INSERT/UPDATE
     *
     * @var array
     */
    private $params = [];

    /**
     * Constructor
     *
     * @param Connector $connector
     */
    public function __construct(Connector $connector)
    {
        $this->connector = $connector;
    }

    /**
     * Return the SQL query string
     */
    public function __toString()
    {
        return $this->buildQuery();
    }

    /**
     * SELECT query
     *
     * @param string $fields
```

```
     * @return self
     */
    public function select(string $fields = '*'): self
    {
        $this->type = 'select';
        $this->fields = $fields;
        return $this;
    }

    /**
     * FROM clause
     *
     * @param string $table
     * @return self
     */
    public function table(string $table): self
    {
        $this->table = $table;
        return $this;
    }

    /**
     * JOIN clause
     *
     * @param string $table
     * @param string $on
     * @param string $type
     * @return self
     */
    public function join(string $key, string $table, string $column,
string $operator = '=', string $type = 'LEFT'): self
    {
        $operator = (in_array($operator,self::operators)) ? $operator :
'=';
        $type = (in_array($type,self::types)) ? $type : 'LEFT';

        $this->join[] = [
            'table' => $table,
            'column' => $column,
            'key' => $key,
            'operator' => $operator,
            'type' => $type
```

```
        ];
        return $this;
    }

    /**
     * WHERE initiator
     *
     * @param string $conjunction
     * @return self
     */
    public function filter(string $conjunction = 'AND'): self
    {
        $conjunction = (in_array($conjunction,self::conjunctions)) ?
$conjunction : 'AND';

        $this->where[] = [
            'conjunction' => $conjunction,
            'clauses' => []
        ];
        $this->filter = count($this->where) - 1;
        return $this;
    }

    /**
     * WHERE clause
     *
     * @param string $column
     * @param mixed  $value
     * @param string $operator
     * @param string $conjunction
     * @return self
     */
    public function where(string $column, $value, string $operator =
null, string $conjunction = 'AND'): self
    {
        $operator = (in_array($operator,self::operators)) ? $operator :
'=';
        $conjunction = (in_array($conjunction,self::conjunctions)) ?
$conjunction : 'AND';

        if(empty($this->where)){
            $this->filter();
        }
```

```
        $this->where[$this->filter]['clauses'][] = [
            'conjunction' => $conjunction,
            'operator' => $operator,
            'column' => $column,
            'value' => $value
        ];
        return $this;
    }

    /**
     * ORDER BY clause
     *
     * @param string $column
     * @param string $direction
     * @return self
     */
    public function order(string $column, string $direction = 'ASC'): self
    {
        $direction = (in_array($direction,self::directions)) ? $direction : 'ASC';

        $this->order[] = [
            'column' => $column,
            'direction' => $direction
        ];
        return $this;
    }

    /**
     * INDEX clause
     *
     * @param string $column
     * @return self
     */
    public function index(string $column): self
    {
        $this->index = $column;
        return $this;
    }

    /**
```

```
     * LIMIT clause
     *
     * @param int $limit
     * @return self
     */
    public function limit(int $limit): self
    {
        $this->limit = $limit;
        return $this;
    }

    /**
     * INSERT query
     *
     * @param array  $data
     * @return self
     */
    public function insert(array $data): self
    {
        $this->type = 'insert';
        $this->values = $data;
        return $this;
    }

    /**
     * UPDATE query
     *
     * @param array  $data
     * @return self
     */
    public function update(array $data): self
    {
        $this->type = 'update';
        $this->values = $data;
        return $this;
    }

    /**
     * DELETE query
     *
     * @return self
     */
    public function delete(): self
```

```
    {
        $this->type = 'delete';
        return $this;
    }

    /**
     * Execute the query and return the result
     *
     * For a SELECT, returns an array of rows.
     * For INSERT, UPDATE, DELETE, returns the number of affected rows.
     *
     * @return mixed
     * @throws Exception
     */
    public function result()
    {
        $sql = $this->buildQuery();
        $result = $this->connector->prepare($sql, $this->params);
        if ($this->type === 'select') {
            $result = $result->get_result();
            $rows = [];
            while ($row = $result->fetch_assoc()) {
                $rows[] = $row;
            }
            return $this->buildRows($rows);
        }
        return $this->connector->affectedRows();
    }

    /**
     * Build the rows array
     *
     * @param array $results
     * @return array
     */
    private function buildRows($results)
    {
        $rows = [];
        foreach ($results as $result){
            $row = [];
            foreach ($result as $key => $value){
                if (strpos($key, 't__') === 0) {
```

```
                    if(!isset($row[substr($key, 3)])){
                        $row[substr($key, 3)] = $value;
                    }
                } else {
                    $parts = explode('__', substr($key, 3));
                    $key = $parts[0];
                    $column = $parts[1];
                    $row[$key] = (!isset($row[$key])) ? [] : $row[$key];
                    $row[$key] = (!is_array($row[$key])) ? [] :
$row[$key];
                    $row[$key][$column] = $value;
                }
            }
            if($this->index && isset($result['t__'.$this->index])){
                $rows[$result['t__'.$this->index]] = $row;
            } else {
                $rows[] = $row;
            }
        }
        return $rows;
    }

    /**
     * Return the ID of the last inserted record
     *
     * @return int
     */
    public function lastId(): int
    {
        return $this->connector->lastId();
    }

    /**
     * Build the SQL query from the given parameters
     *
     * @return string
     */
    private function buildQuery(): string
    {
        $this->params = [];
        switch ($this->type) {
            case 'select':
                $sql = "SELECT {$this->buildFields()} FROM
```

```
`{$this->table}` AS t";
                if (!empty($this->join)) {
                    $sql .= ' ' . $this->buildJoin();
                }
                if (!empty($this->where)) {
                    $sql .= ' WHERE ' . $this->buildWhere();
                }
                if (!empty($this->order)) {
                    $sql .= ' ORDER BY ' . $this->buildOrder();
                }
                if ($this->limit) {
                    $sql .= " LIMIT {$this->limit}";
                }
                return $sql;

            case 'insert':
                $columns = array_keys($this->values);
                foreach ($columns as $key => $column) {
                    $columns[$key] = "`{$column}`";
                }
                $columns = implode(',', $columns);
                $values = implode(',', array_map([$this, 'addParam'],
array_values($this->values)));
                return "INSERT INTO `{$this->table}` ({$columns}) VALUES
({$values})";

            case 'update':
                $sets = [];
                foreach ($this->values as $col => $val) {
                    $sets[] = "`{$col}` = " . $this->addParam($val);
                }
                $sql = "UPDATE `{$this->table}` SET " . implode(', ',
$sets);
                if (!empty($this->where)) {
                    $sql .= ' WHERE ' . $this->buildWhere();
                }
                return $sql;

            case 'delete':
                $sql = "DELETE FROM `{$this->table}`";
                if (!empty($this->where)) {
                    $sql .= ' WHERE ' . $this->buildWhere();
```

```
                }
                return $sql;
            }

            throw new Exception("Invalid query type: {$this->type}");
        }

        /**
         * Build the Fields clause
         *
         * @return string
         */
        private function buildFields(): string
        {
            if($this->fields == '*'){
                $fields = [];
                $columns = $this->connector->describe($this->table);
                foreach ($columns as $column) {
                    $fields[] = "t.`{$column['Field']}` AS
`t__{$column['Field']}`";
                }
                foreach ($this->join as $join) {
                    $columns = $this->connector->describe($join['table']);
                    foreach ($columns as $column) {
                        $fields[] = "j__{$join['key']}.`{$column['Field']}`
AS `j__{$join['key']}__{$column['Field']}`";
                    }
                }
            } else {
                $fields = explode(',', $this->fields);
                foreach ($fields as $key => $field) {
                    $field = trim($field);
                    if(strpos($field, '.') !== false){
                        $parts = explode('.', $field);
                        $joint = $parts[0];
                        $column = $parts[1];
                        $fields[$key] = "`j__{$joint}`.`{$column}` AS
`j__{$joint}__{$column}`";
                    } else {
                        $fields[$key] = "t.`{$field}` AS `t__{$field}`";
                    }
                }
            }
```

```
        return implode(', ', $fields);
    }

    /**
     * Build the JOIN clause
     *
     * @return string
     */
    private function buildJoin(): string
    {
        $clauses = [];
        foreach ($this->join as $join) {
            $clauses[] = "{$join['type']} JOIN `{$join['table']}` AS
`j__{$join['key']}` ON t.`{$join['key']}` {$join['operator']}
`j__{$join['key']}`.`{$join['column']}`";
        }
        return implode(' ', $clauses);
    }

    /**
     * Build the WHERE clause
     *
     * @return string
     */
    private function buildWhere(): string
    {
        $filters = '';
        foreach ($this->where as $filter => $where) {
            $clauses = [];
            foreach ($where['clauses'] as $i => $condition) {
                $conjunction = $condition['conjunction'];
                if($this->type === 'select'){
                    $col = (strpos($condition['column'], '.') !== false)
? "`".str_replace('.','`.`',$condition['column'])."`" :
"t.`{$condition['column']}`";
                } else {
                    $col = "`{$condition['column']}`";
                }
                $op = $condition['operator'];
                $val = (in_array($op, ['IS NULL', 'IS NOT NULL'])) ? ''
: $this->addParam($condition['value']);
                $clause = "{$col} {$op} {$val}";
```

```
                $clauses[] = ($i > 0) ? "{$conjunction} {$clause}" :
$clause;
            }
            $filters .= ($filter > 0) ? " {$where['conjunction']} " :
'';
            $filters .= "(".implode(' ', $clauses).")";
        }
        return $filters;
    }

    /**
     * Build the ORDER BY clause
     *
     * @return string
     */
    private function buildOrder(): string
    {
        $clauses = [];
        foreach ($this->order as $order) {
            $clauses[] = "`{$order['column']}` {$order['direction']}";
        }
        return implode(', ', $clauses);
    }

    /**
     * Add a parameter to the list and return a placeholder
     *
     * @param mixed $value
     * @return string
     */
    private function addParam($value): string
    {
        if (is_array($value)) {
            $value = json_encode($value, JSON_UNESCAPED_SLASHES |
JSON_PRETTY_PRINT);
        }
        $this->params[] = $value;

        return '?';
    }
}
```

La classe **Query** permet de construire des requêtes SQL de façon structurée et chaînable. Une fois la requête bâtie, un appel à `→result()` l'exécute. Par exemple :

```
$db = new Database();
$data = $db->query()
    ->select('*')
    ->table('users')
    ->filter()
    ->where('username', 'jdoe')
    ->result();
```

LES CLASSES SCHEMA ET DEFINITION

Sources: Schema.php

src/Schema.php

```
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use LaswitchTech\Core\Definition;
use LaswitchTech\Core\Connector;
use Exception;

class Schema {

    /**
     * @var Connector
```

```
     */
    private $connector;

    /**
     * Definition path
     *
     * @var string
     */
    private $path;

    /**
     * Table name
     *
     * @var string
     */
    private $table;

    /**
     * Definitions
     *
     * @var array
     */
    private $definitions = [];

    /**
     * Table engine
     *
     * @var string
     */
    private $engine = 'InnoDB';

    /**
     * Table charset
     *
     * @var string
     */
    private $charset = 'utf8mb4';

    /**
     * Table collation
     *
     * @var string
     */
```

```
    private $collation = 'unicode_ci';

    /**
     * Constructor
     *
     * @param Connector $connector
     */
    public function __construct(Connector $connector)
    {
        global $CONFIG;
        $this->path = $CONFIG->root() . '/Definition';

        $this->connector = $connector;
    }

    /**
     * Describe a table
     *
     * @return mixed
     */
    public function describe()
    {
        $definitions = [];
        foreach ($this->definitions as $column => $definition) {
            $definitions[] = $definition->define();
        }
        return $definitions;
    }

    /**
     * DEFINE the table
     *
     * @param string $table
     * @return self
     */
    public function define(string $table): self
    {
        $this->table = $table;
        $path = $this->path . DIRECTORY_SEPARATOR . $table . '.map';

        // If the definition file does not exist, generate the
definition
```

```
        if (!file_exists($path)) {

            // Load the table definition
            $definitions = $this->connector->describe($table);

            // Retrieve the table engine, charset and collation
            $sql = "SHOW TABLE STATUS LIKE '{$table}'";
            $result = $this->connector->query($sql);
            $row = $result->fetch_assoc();

            // Set the table engine
            $this->engine = $row['Engine'];

            // Set the table charset and collation
            $Collation = explode('_', $row['Collation']);
            $this->charset = $Collation[0];
            $this->collation =
str_replace($this->charset.'_','',$row['Collation']);
        } else {

            // Retrieve the definition from the file
            $definitions = json_decode(file_get_contents($path), true);

            // Loop through the definition
            foreach ($definitions as $key => $definition) {
                if(in_array($key, ['engine', 'charset', 'collation'])) {
                    $this->{$key} = $definition;
                    unset($definitions[$key]);
                }
            }
        }

        // Load the definition
        foreach ($definitions as $key => $definition) {
            $this->column($definition['Field'], $definition);
        }

        // If the definition file does not exist, create it
        if (!file_exists($path)) {

            // Save the definition
            $this->save();
        }
```

```
        return $this;
    }

    /**
     * Get a column's Definition
     *
     * @param string $column
     * @param array $definition
     * @return Definition
     */
    public function column(string $column, array $definition = []):
Definition
    {
        if(!isset($this->definitions[$column])) {
            $this->definitions[$column] = new Definition($column,
$definition);
        }
        return $this->definitions[$column];
    }

    /**
     * Rename a column
     *
     * @param string $column
     * @param string $newName
     * @return self
     */
    public function rename(string $column, string $newName): self
    {
        if(isset($this->definitions[$column])) {
            $this->definitions[$newName] = $this->definitions[$column];
            unset($this->definitions[$column]);
            $this->definitions[$newName]->rename($newName);
        }
        // $sql = "ALTER TABLE `{$this->table}` CHANGE `{$column}`
`{$newName}` {$this->definitions[$newName]->define()['Type']}";
        // $this->connector->query($sql);
        return $this;
    }

    /**
```

```
     * Remove a column
     *
     * @param string $column
     * @return self
     */
    public function remove(string $column): self
    {
        if(isset($this->definitions[$column])) {
            unset($this->definitions[$column]);
        }
        return $this;
    }

    /**
     * Save the definition to a file
     *
     * @return self
     */
    public function save(): self
    {
        $definitions = [];
        foreach ($this->definitions as $definition) {
            $definitions[] = $definition->define();
        }
        $definitions['engine'] = $this->engine;
        $definitions['charset'] = $this->charset;
        $definitions['collation'] = $this->collation;
        $path = $this->path . DIRECTORY_SEPARATOR . $this->table .
'.map';
        if(!is_dir($this->path)) { mkdir($this->path, 0777, true); }
        file_put_contents($path, json_encode($definitions,
JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES));
        return $this;
    }

    /**
     * Compare the in-memory definition with the actual DB structure.
     *
     * If $asQueries = false, return a descriptive array of differences.
     * If $asQueries = true, return an array of SQL statements that
would
     * apply the changes needed to sync the DB to the in-memory
definitions.
```

```
     *
     * @param bool $asQueries
     * @return array
     */
    public function compare(bool $asQueries = false): array
    {
        // Make sure we have a table name
        if (empty($this->table)) {
            return [];
        }

        // Get columns from DB
        $dbCols = $this->connector->describe($this->table);
        $dbMap = [];
        foreach ($dbCols as $col) {
            $dbMap[$col['Field']] = $col;
        }

        // Get columns from memory
        $memoryMap = [];
        foreach ($this->definitions as $definitionObj) {
            $arr = $definitionObj->define();
            $memoryMap[$arr['Field']] = $arr;
        }

        // We will accumulate a list of changes (descriptive or queries)
        $changes = [];

        // 1) Handle Renamed Columns
        //    If in-memory column has "Previously" => we interpret that
as "rename oldName -> newName"
        //    if oldName actually exists in $dbMap, we know it's a
rename.
        //    Then we can remove oldName from $dbMap to avoid also
dropping it below.
        foreach ($memoryMap as $newName => $memDef) {
            if (!empty($memDef['Previously']) &&
isset($dbMap[$memDef['Previously']])) {
                $oldName = $memDef['Previously'];
                if (isset($dbMap[$oldName]) &&
!isset($changes[$newName])) {
                    // It's truly a rename from $oldName to $newName
```

```
                    if ($asQueries) {
                        $changes[$newName] =
$this->buildRenameQuery($oldName, $newName, $memDef);
                    } else {
                        $changes[$newName] = [
                            'action'   => 'rename',
                            'oldName'  => $oldName,
                            'newName'  => $newName,
                            'details'  => $memDef
                        ];
                    }
                    // We handle the rename by removing the oldName from
$dbMap
                    // and also removing $newName from $memoryMap so we
don't handle it again
                    unset($dbMap[$oldName]);
                    // (We won't remove from $memoryMap here, because we
need it for next steps—some prefer to do so, but you can also mark it
somehow.)
                }
            }
        }

        // 2) Handle Added Columns
        //    If a column is in memoryMap but not in $dbMap => it's a
new column
        foreach ($memoryMap as $colName => $memDef) {
            if (!isset($dbMap[$colName]) && !isset($changes[$colName]))
{
                // We skip columns that are flagged as a rename
                // if $memDef['Previously'] was set but didn't match a
DB col, this might be a rename from a non-existing col
                // We'll treat it as "add" anyway
                if ($asQueries) {
                    $changes[$colName] =
$this->buildAddColumnQuery($colName, $memDef);
                } else {
                    $changes[$colName] = [
                        'action'  => 'add',
                        'column'  => $colName,
                        'details' => $memDef
                    ];
                }
```

```
            }
        }

        // 3) Handle Dropped Columns
        //    If a column is in DB but not in memory => it's removed
        //    (unless it's renamed, which we handled earlier by
unsetting from $dbMap)
        foreach ($dbMap as $colName => $dbDef) {
            if (!isset($memoryMap[$colName]) &&
!isset($changes[$colName])) {
                if ($asQueries) {
                    $changes[$colName] = "ALTER TABLE `{$this->table}`
DROP COLUMN `{$colName}`";
                } else {
                    $changes[$colName] = [
                        'action' => 'drop',
                        'column' => $colName,
                        'dbDef'  => $dbDef
                    ];
                }
            }
        }

        // 4) Handle Modified Columns
        //    If a column is both in DB and in memory (and not renamed),
check differences (type, null, default, etc.)
        //    Then build a "modify" or "change" query if there's a
difference
        foreach ($memoryMap as $colName => $memDef) {
            if (isset($dbMap[$colName]) && !isset($changes[$colName])) {
                // Compare details
                $dbDef = $dbMap[$colName];
                // We'll do a simple comparison on "Type", "Null",
"Default", "Extra", "Key"
                $diff = $this->compareColumnDefs($dbDef, $memDef);
                if (!empty($diff)) {
                    // We have differences
                    if ($asQueries) {
                        // A "MODIFY COLUMN" or "CHANGE COLUMN"
statement can be used.
                        // Typically use "MODIFY" if we keep the same
col name
```

```
                        // We'll build a "MODIFY" statement
                        $changes[$colName] =
$this->buildModifyColumnQuery($colName, $memDef);
                    } else {
                        $changes[$colName] = [
                            'action' => 'modify',
                            'column' => $colName,
                            'dbDef'  => $dbDef,
                            'memDef' => $memDef,
                            'diff'   => $diff
                        ];
                    }
                }
            }
        }

        return $changes;
    }

    /**
     * Compare two column definitions (DB vs memory).
     * Return an array of differences or an empty array if none.
     *
     * For simplicity, we only compare keys we care about:
     * 'Type', 'Null', 'Default', 'Extra', 'Key'.
     */
    private function compareColumnDefs(array $dbDef, array $memDef):
array
    {
        $keysToCheck = array_keys($dbDef);
        $diff = [];
        foreach ($keysToCheck as $key) {
            // Normalize or parse if needed, e.g. 'int(11)' vs 'int(10)'
            // For now, let's do a plain string compare
            $dbVal  = isset($dbDef[$key])  ? $dbDef[$key]  : null;
            $memVal = isset($memDef[$key]) ? $memDef[$key] : null;
            if ($dbVal != $memVal) {
                $diff[$key] = ["db"=>$dbVal, "mem"=>$memVal];
            }
        }
        return $diff;
    }
```

```
    /**
     * Build the SQL to rename a column.
     * e.g. ALTER TABLE tableName CHANGE oldName newName newDefinition
     */
    private function buildRenameQuery(string $oldName, string $newName,
array $memDef): string
    {
        $colDefSQL = $this->buildColumnSQL($memDef, $newName);
        return "ALTER TABLE `{$this->table}` CHANGE `{$oldName}`
`{$newName}` {$colDefSQL}";
    }

    /**
     * Build the SQL to add a new column.
     * e.g. ALTER TABLE tableName ADD COLUMN colName colDefinition
     */
    private function buildAddColumnQuery(string $colName, array
$memDef): string
    {
        $colDefSQL = $this->buildColumnSQL($memDef, $colName);
        return "ALTER TABLE `{$this->table}` ADD COLUMN {$colDefSQL}";
    }

    /**
     * Build the SQL to modify an existing column (same name, changed
type, etc).
     * e.g. ALTER TABLE tableName MODIFY COLUMN colName colDefinition
     */
    private function buildModifyColumnQuery(string $colName, array
$memDef): string
    {
        $colDefSQL = $this->buildColumnSQL($memDef, $colName);
        return "ALTER TABLE `{$this->table}` MODIFY COLUMN
{$colDefSQL}";
    }

    /**
     * Build the column definition snippet, e.g.
     *   `age` int(11) NOT NULL DEFAULT '0' AUTO_INCREMENT
     *
     * If the column is 'NULL' then we do "NULL" instead of "NOT NULL".
     */
```

```
    private function buildColumnSQL(array $memDef, string $colName): string
    {
        // We'll need:
        // - column name
        // - type
        // - null vs not null
        // - default
        // - extra (auto_increment / on update CURRENT_TIMESTAMP)
        // Possibly also Key, but in MySQL you'd typically do PK or unique in separate statements,
        // or in the create statement. For an "ALTER" approach, we might skip 'Key' here.

        $type = $memDef['Type'] ?? 'varchar(255)';

        // Null / not null
        $null = (isset($memDef['Null']) && $memDef['Null'] === 'YES') ? "NULL" : "NOT NULL";

        // Default
        $defaultSQL = "";
        if(!in_array($type, ['tinytext', 'text', 'mediumtext', 'longtext', 'tinyblob', 'blob', 'mediumblob', 'longblob'])) {
            if (array_key_exists('Default', $memDef)) {
                $defaultVal = $memDef['Default'];
                if ($defaultVal === 'CURRENT_TIMESTAMP') {
                    $defaultSQL = "DEFAULT CURRENT_TIMESTAMP";
                } else {
                    if ($defaultVal === null && $null === "NULL") {
                        $defaultSQL = "DEFAULT NULL";
                    } elseif ($defaultVal !== null) {
                        $escaped = addslashes($defaultVal);
                        $defaultSQL = "DEFAULT '{$escaped}'";
                    }
                }
            }
        }

        // Extra
        $extra = "";
        if (!empty($memDef['Extra'])) {
            // e.g. "auto_increment" or "on update CURRENT_TIMESTAMP"
```

```
            $extra = strtoupper($memDef['Extra']);
        }

        return "`{$colName}` {$type} {$null}"
               . (empty($defaultSQL) ? "" : " {$defaultSQL}")
               . (empty($extra) ? "" : " {$extra}");
    }

    /**
     * Build the CREATE TABLE SQL, e.g.
     *  CREATE TABLE `tableName` (
     *
     * @return string
     */
    private function buildCreate(): string
    {
        $columnLines = [];
        foreach ($this->definitions as $definition) {
            $columnLines[] =
$this->buildColumnSQL($definition->define(),
$definition->define()['Field']);
        }

        // Check if we have primary keys
        $primaryKeys = [];
        foreach ($this->definitions as $definition) {
            if (isset($definition->define()['Key']) &&
$definition->define()['Key'] === 'PRI') {
                $primaryKeys[] = $definition->define()['Field'];
            }
        }

        // If we have primary keys, add a line for them
        if (!empty($primaryKeys)) {
            $pk = "`" . implode("`,`", $primaryKeys) . "`";
            $columnLines[] = "PRIMARY KEY ($pk)";
        }

        // For demonstration, assume InnoDB engine
        // In real usage, you might store the engine in definition or
config
        $columnsSQL = implode(",\n  ", $columnLines);
```

```
        $sql = "CREATE TABLE `{$this->table}` (\n  {$columnsSQL}\n)
ENGINE={$this->engine} DEFAULT CHARSET={$this->charset}
COLLATE={$this->collation}";
        return $sql;
    }

    /**
     * Drop the table if it exists
     *
     * @return self
     * @throws Exception
     */
    public function drop(): self
    {
        if (empty($this->table)) {
            throw new Exception("No table defined.");
        }
        $sql = "DROP TABLE IF EXISTS `{$this->table}`";
        $this->connector->query($sql);

        return $this;
    }

    /**
     * Truncate the table
     *
     * @return self
     * @throws Exception
     */
    public function truncate(): self
    {
        if (empty($this->table)) {
            throw new Exception("No table defined.");
        }
        $sql = "TRUNCATE TABLE `{$this->table}`";
        $this->connector->query($sql);

        return $this;
    }

    /**
     * Update the table
     *
```

```
     * @return self
     */
    public function update(): self
    {
        $changes = $this->compare(true);
        foreach ($changes as $change) {
            $this->connector->query($change);
        }
        return $this;
    }

    /**
     * Create the table
     *
     * @return self
     */
    public function create(): self
    {
        $sql = $this->buildCreate();
        $this->connector->query($sql);
        return $this;
    }

    /**
     * Return if the table exists
     *
     * @return bool
     */
    public function exists(): bool
    {
        $sql = "SHOW TABLES LIKE '{$this->table}'";
        $result = $this->connector->query($sql);
        return $result->num_rows > 0;
    }
}
```

Sources: Definition.php

src/Definition.php

```php
<?php

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

// Declaring namespace
namespace LaswitchTech\Core;

// Import additionnal class into the global namespace
use Exception;

class Definition {

    /**
     * List of default length for types
     *
     * @var array
     */
    const lengths = [
        // Numeric
        'int' => 11,
        'tinyint' => 4,
        'smallint' => 6,
        'mediumint' => 9,
        'bigint' => 20,
        // -
        'decimal' => 10,
        'float' => 12,
        'double' => 22,
        'real' => 12,
        // -
        'bit' => 1,
        'boolean' => 1,
        'serial' => 10,

        // Date and Time
        'date' => 0,
```

```
            'datetime' => 0,
            'timestamp' => 0,
            'time' => 0,
            'year' => 4,

            // String
            'char' => 255,
            'varchar' => 255,
            // -
            'tinytext' => 0,
            'text' => 0,
            'mediumtext' => 0,
            'longtext' => 0,
            // -
            'binary' => 255,
            'varbinary' => 255,
            // -
            'tinyblob' => 0,
            'blob' => 0,
            'mediumblob' => 0,
            'longblob' => 0,
            // -
            'enum' => 0,
            'set' => 0,

            // Spatial
            'geometry' => 0,
            'point' => 0,
            'linestring' => 0,
            'polygon' => 0,
            'multipoint' => 0,
            'multilinestring' => 0,
            'multipolygon' => 0,
            'geometrycollection' => 0,

            // JSON
            'json' => 0,
        ];

        /**
         * Column name
         *
```

```
     * @var string
     */
    private $column;

    /**
     * Table definition
     *
     * @var array
     */
    private $definition = [
        "Field" => null,
        "Type" => "varchar(255)",
        "Null" => "YES",
        "Key" => "",
        "Default" => null,
        "Extra" => ""
    ];

    /**
     * Constructor
     *
     * @param string $column
     * @param array $definition
     */
    public function __construct(string $column, array $definition = [])
    {
        $this->column = $column;
        foreach($definition as $key => $value) {
            $this->definition[$key] = $value;
        }
        $this->definition['Field'] = $column;
    }

    /**
     * Define the column
     *
     * @return array
     */
    public function define(): array
    {
        return $this->definition;
    }
```

```php
    /**
     * Set the column type
     *
     * @param string $type
     * @param int $length
     * @return self
     */
    public function type(string $type, int $length = 0): self
    {
        $length = ($length > 0) ? $length : self::lengths[$type];
        $this->definition['Type'] = $type . ($length > 0 ? "($length)" :
"");
        return $this;
    }

    /**
     * Set the column as nullable
     *
     * @param bool $isNull
     * @return self
     */
    public function nullable(bool $isNull = true): self
    {
        $this->definition['Null'] = ($isNull) ? "YES" : "NO";
        return $this;
    }

    /**
     * Set the column default value
     *
     * @param string $value
     * @return self
     */
    public function default(string $value = null): self
    {
        $this->definition['Default'] = $value;
        return $this;
    }

    /**
     * Set the column as primary key
     *
```

```
     * @param bool $isPrimary
     * @return self
     */
    public function primary(bool $isPrimary = true): self
    {
        $this->definition['Key'] = ($isPrimary) ? "PRI" : "";
        return $this;
    }

    /**
     * Set the column as unique
     *
     * @param bool $isUnique
     * @return self
     */
    public function unique(bool $isUnique = true): self
    {
        $this->definition['Key'] = ($isUnique) ? "UNI" : "";
        return $this;
    }

    /**
     * Set the column as auto increment
     *
     * @param bool $isAutoIncrement
     * @return self
     */
    public function autoIncrement(bool $isAutoIncrement = true): self
    {
        $this->definition['Extra'] = ($isAutoIncrement) ?
"auto_increment" : "";
        return $this;
    }

    /**
     * Set on update current timestamp
     *
     * @param bool $isOnUpdate
     * @return self
     */
    public function onUpdate(bool $isOnUpdate = true): self
    {
        $this->definition['Extra'] = ($isOnUpdate) ? "on update
```

```
CURRENT_TIMESTAMP" : "";
        return $this;
    }

    /**
     * Rename the column
     *
     * @param string $name
     * @return self
     */
    public function rename(string $name): self
    {
        $this->definition['Field'] = $name;
        $this->definition['Previously'] = $this->column;
        $this->column = $name;
        return $this;
    }
}
```

Ces classes vous permettent de gérer la structure des tables et leurs colonnes de manière plus programmatique. Vous pouvez :

- Générer (ou lire) des définitions de colonnes
- Suivre les modifications du schéma
- Comparer ou renommer des colonnes
- Créer de nouvelles tables ou mettre à jour celles existantes

CONCLUSION

Nous avons maintenant implémenté plusieurs fonctionnalités essentielles dans notre framework PHP :

- **Protection CSRF** pour sécuriser votre application contre les soumissions de formulaires malveillants
- **Module API** pour gérer le routage, le traitement des requêtes et, plus tard, l'authentification et l'autorisation

- **Gestion de la base de données** grâce à des connecteurs (ex. MySQL), un générateur de requêtes enchaînables (**Query**) et un système de gestion de schéma (**Schema**) pour définir et mettre à jour les tables

Grâce à ces éléments, notre framework peut maintenant gérer une sécurité de base, des opérations de données robustes et des structures modulaires pour les endpoints. Dans les prochains volets, nous allons approfondir les mécanismes d'authentification, discuter de routage avancé ou d'architectures de plugins, et continuer à peaufiner notre framework pour qu'il soit à la fois extensible et sécurisé.

DÉPÔT GITHUB

v0.0.11

TAGSGENERALCORE-FRAMEWORK-FRFRAMEWORKCOREMODULARITYMODULARMODULELIGHTWEIGHTAPPLICATIONPHPCAKEPHPSYMFONYLEARNLEANINGMAINTAINABILITYMAINTAIN

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

View the discussion thread.

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

Permanent link:
https://laswitchtech.com/fr/blog/2025/02/03/let-s-talk-building-a-modular-php-framework-part-3

Last update: **2026/02/16 13:31**