API Rest pour l'administration système

TD Création d'une API en PHP - Mastère Systèmes - Réseaux et Cloud Computing - 18/12/2024

Création d’une API Rest en PHP

Objectifs

- À partir d'une base de données SQLite. Votre mission est de créer une API basique qui permette de lire, d'ajouter, de modifier et de supprimer des données.
- Vous devez utiliser PHP et PDO pour accéder à la base de données.
- Vous devez utiliser les verbes HTTP pour effectuer les opérations CRUD.
- Vous devez utiliser le format JSON pour les échanges de données.
- Vous devez utiliser un système de routage pour gérer les différentes routes de l'API.
- Vous devez utiliser un système de gestion des erreurs pour renvoyer des messages d'erreur en cas de problème.

Nous alons voir dans ce TD comment procéder.

Prérequis

Avoir le serveur web PHP Apache installé sur votre système (Fourni avec XAMPP si vous avez installé XAMPP).

Avoir un éditeur de texte ou un IDE pour écrire le code.

Avoir une version de PHP de préférence supérieure à 8.1, pour pouvoir utiliser les dernières fonctionnalités de PHP.

Nous utiliserons Postman pour tester notre requêtes. S’assurer donc de l’avoir installé sur votre machine.

Utilisation de Docker (facultatif)

Si vous ne voulez pas installer PHP et Apache sur votre système, vous pouvez utiliser Docker pour créer un conteneur avec PHP et Apache.

Pour cela vous pouvez utiliser le dockerfile suivant :

```
FROM php:8.2-apache
RUN a2enmod rewrite # Activer le module rewrite d'Apache (nous en aurons besoin pour le routage)
```

Ainsi que le docker-compose suivant :

```
version: '3'
services:
  php-apache:
    build: .
    dockerfile: Dockerfile
    ports:
      - "8080:80"
    volumes:
      - .:/var/www/html
```

Nous utiliserons une base de données SQLite pour stocker les données.

La base de données sqlite est fournie dans les sources du projet.

Let’s Go !

Nous sommes prêts à commencer à coder.

Créez un dossier (dans votre répertoire `htdocs` si vous utilisez `XAMPP`), nous allons commencer à coder notre API.

1. Système de routage

Système de routage

Créer votre fichier `index.php` dans un dossier `api` de votre projet. Vérifiions simplement que notre serveur fonctionne.

`api/index.php`

```php
<?php

echo 'Hello World';
```

Utilisez votre navigateur pour accéder à `http://localhost:8080/api/` (adaptez le numéro de port en fonction de votre serveur) et vérifiez que vous voyez `Hello World`.

Système de routage

Problème !

Nous avons pour objectif d'accéder aux ressources sur les urls suivantes :

- GET `http://localhost:8080/api/users/` pour récupérer tous les utilisateurs
- GET `http://localhost:8080/api/users/1` pour récupérer l'utilisateur avec l'id 1
- POST `http://localhost:8080/api/users` pour créer un utilisateur
- PUT `http://localhost:8080/api/users/1` pour mettre à jour l'utilisateur avec l'id 1

Comment faire pour que notre fichier `index.php` puisse gérer ces différentes routes ?

Solution: Utiliser l'URL Rewriting !

Système de routage

Pour pouvoir faire cela sous apache nous allons créer un fichier `.htaccess` qui s'occupera d'intercepter les requêtes et de les rediriger vers notre fichier `index.php`.

Créez donc un fichier `.htaccess` dans le dossier `/api`.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-l
RewriteRule . index.php [L]
```

Plus d'informations sur les fichiers htaccess d'Apache: https://httpd.apache.org/docs/2.4/howto/htaccess.htm

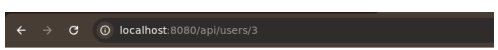
Ainsi toutes les url qui ne correspondent pas à un fichier existant seront redirigées vers `index.php`.

Système de routage

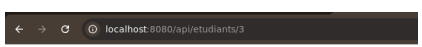
Pour en avoir la preuve, et aussi pour récupérer les ressources de l'url, nous allons modifier notre fichier `index.php` pour afficher les paramètres de l'url.

```php
<?php
$path = parse_url($_SERVER["REQUEST_URI"], PHP_URL_PATH); // récupéràtion de l'url

// décomposition de l'url
$parts = explode("/", $path);

$resource = $parts[2];

$id = $parts[3] ?? null;

echo "La ressource demandée est $resource, avec l'identifiant $id";
```

Ainsi nous pouvons bien intercepter toutes les requêtes et récupérer les ressources et les identifiants:



La ressource demandée est users, avec l'identifiant 3



La ressource demandée est etudiants, avec l'identifiant 3

Système de routage

Pour obtenir le méthode on peut utiliser la variable `$_SERVER['REQUEST_METHOD']`.

Ainsi on peut compléter notre fichier `index.php` pour répondre uniquement à la ressource voulue:

```php
<?php
$path = parse_url($_SERVER["REQUEST_URI"], PHP_URL_PATH); // récupéràtion de l'url

// décomposition de l'url
$parts = explode("/", $path);

$resource = $parts[2];

$id = $parts[3] ?? null;

switch($resource){
  case 'users':
    // il faudra implémenter les actions pour les utilisateurs
    // nous verrons cela plus tard
  break;
  default:
    http_response_code(404); # Définition du code de statut HTTP
    echo json_encode(["message" => "Resource not found"]);
    exit;

}
```

Controller

Nous allons maintenant créer une classe de type "Controller" qui s'occupera pour chaque ressource de gérer les différentes méthodes HTTP.

Créez un dossier `src` dans votre projet et créez un fichier `UserController.php`.

`src/UserController.php`

```php
class UserController
{
  public function processRequest(string $method, string $id = null)
  {
    if ($id){
      // si un id est détecté, on effectue certaines actions
      switch($method){
          // ...
      }

    }else {
      // si il n'y a pas d'id on effectue les actions de collection
      switch ($method) {
        // ...
      }
    }
```

```php
class UserController
{
  public function processRequest(string $method, string $id = null)
  {
    if ($id){
      // si un id est détecté, on effectue certaines actions
      switch($method){
          case 'GET':
            echo "méthode GET appelée sur l'utilisateur $id";
            break;
          default:
            $this->methodNotAllowed('GET', 'POST');
      }

    }else {
      // si il n'y a pas d'id on effectue les actions de collection
      switch ($method) {
        case 'GET':
          echo 'méthode GET appelée';
          break;
        case 'POST':
          echo 'méthode POST appelée';
          break;
        default:
          $this->methodNotAllowed('GET', 'POST');
      }
    }
```

```php
    private function responseMethodNotAllowed(string $allowedMethods): void {
        http_response_code(405); // METHOD NOT ALLOWED RESPONSE CODE
        header("Allow: $allowedMethods");
    }
}
```

Pour l’instant nous avons simplement un système qui permet de retourner une erreur 405 si la méthode HTTP n’est pas autorisée.

Plus tard nous implémenterons les actions CRUD pour chaque méthode.

Connection à la base de données

Il est temps de se connecter à la base de données pour récupérer les données.

Nous allons séparer cette logique dans une classe spécifique `Database.php`.

Créez donc un fichier `Database.php` dans le dossier `src`.

```php
<?php
class Database {

    public function getConnection(): PDO {
        $dsn = "sqlite:" . __DIR__ . "/../users.sqlite";
        return new PDO($dsn , null , null, [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION // indique que nous voulons une exception en cas d'erreur
        ]);
    }
}
```

La connection se fait via un object natif php appelé `PDO` qui permet de se connecter à de nombreuses bases de données. (voir la documentation officielle de PHP pour plus d'informations sur PDO: https://www.php.net/manual/fr/book.pdo.php)

Connection à la base de données

Pour une meilleure organisation de notre code, nous allons créer une classe `Gateway` qui s'occupera de gérer les requêtes SQL.

Créez un fichier `Gateway.php` dans le dossier `src`.

```php
class UserGateway {
    private PDO $connection;

    public function __construct(private PDO $connection) {
      // le constructeur permet de passer l'objet PDO qui sera utilisé pour les requêtes
    }

    public function getAll(): array {
      // ce sera la méthode appelée lors d'une requête GET sur /users
    }

    public function getById(string $id)  {
      // ce sera la méthode appelée lors d'une requête GET sur /users/{id}
    }

    public function insert(array $data): void {
      // ce sera la méthode appelée lors d'une requête POST sur /users
```

Connection à la base de données

Voici comment nous pouvons récupérer tous les utilisateurs de la base de données.

```
...
  public function getAll(): array {
      $statement = $this->connection->query("SELECT * FROM users");
      // Si on voulait tout retourner d'un coup on pourrait utiliser fetchAll
      // return $statement->fetchAll(PDO::FETCH_ASSOC);

      // mais ici nous pouvons adapater les données
      $data = [];
      while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
          $data[] = [
              "id" => $row["id"],
              "first_name" => $row["first_name"],
              "last_name" => $row["first_name"],
              "email" => $row["email"],
              "avatar" => $row["avatar"]
          ];
      }
      return $data;
  }
```

Revenons à notre fichier d'entrée index.php

Maintenant que nous avons nos classes `Database` et `UserGateway` nous pouvons les utiliser dans notre fichier `index.php`. La logique se passe dans le `switch($resource)`

```php
switch ($resource){
    case "users":
        $database = new Database();
        $userGateway = new UserGateway($database->getConnection());
        $controller = new UserController($userGateway);
        $controller->processRequest($_SERVER["REQUEST_METHOD"], $id);
        break;
    default:
        http_response_code(404);
        echo json_encode(["message" => "Not Found"]);
        exit;
}
```

Mais ceci ne vas pas marcher tout de suite car comme nous utilisons des classes d'autres fichiers, nous allons devoir les inclure dans notre fichier `index.php`.

Pour faire cela nous allons utiliser l'autoloader du gestionnaire de paquets de PHP (composer).

Utilisation de Composer

Composer est un gestionnaire de dépendances pour PHP. Il permet de gérer les dépendances de votre projet et de les installer automatiquement. Pour notre cas, nous allons l'utiliser pour générer un autoloader qui va inclure automatiquement les classes de notre projet.

Pour commencer, installez composer sur votre machine en suivant les instructions sur le site officiel: https://getcomposer.org/

Ensuite, créez un fichier `composer.json` à la racine de votre projet.

```
{
    "autoload": {
        "psr-4": {
            "": "src/"
        }
    }
}
```

Ensuite exécutez la commande `composer dump-autoload` pour générer l'autoloader.

Utilisation de Composer

Un dossier `vendor` a été créé à la racine de votre projet. Il contient l'autoloader généré par composer.

Maintenant que nous avons notre autoloader, nous pouvons l'utiliser dans notre fichier `index.php`.

```php
require dirname(__DIR__) . '/vendor/autoload.php';
```

`dirname` permet de remonter d'un niveau pour accéder au dossier `vendor` qui contient l'autoloader.
`__DIR__` est une constante magique de PHP qui contient le chemin du dossier courant.

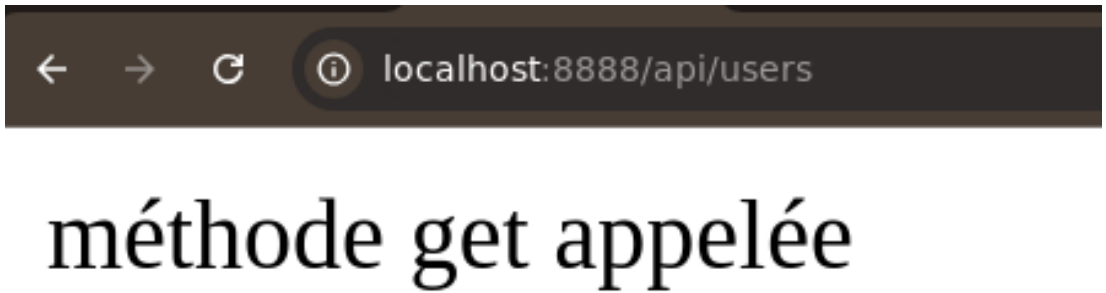
Utilisation de Composer

Désormais nous pouvons utiliser nos classes dans notre fichier index.php sans avoir à les inclure manuellement.

```php
require_once dirname(__DIR__) . "/vendor/autoload.php";

$path = parse_url($_SERVER["REQUEST_URI"], PHP_URL_PATH);

// décomposition de l'url
$parts = explode("/", $path);

$resource = $parts[2] ?? null;
$id = $parts[3] ?? null;

switch ($resource){
    case "users":
        $database = new Database();
        $userGateway = new UserGateway($database->getConnection());
        $controller = new UserController($userGateway);
        $controller->processRequest($_SERVER["REQUEST_METHOD"], $id);
        break;
    default:
        http_response_code(404);
        echo json_encode(["message" => "Not Found"]);
```

vérification

Nous pouvons faire un test en faisant une requête GET sur `http://localhost:8080/api/users/` :



Refactorisation

Pour être plus rigoureux sur la gestion des erreurs, et des exceptions nous allons créer une classe spécifique qui nous permettra de renvoyer à coup sur du json.

Créez un fichier `ErrorHandler.php` dans le dossier `src`.

```php
<?php

class ErrorHandler {
    public static function handleError( int $errno, string $errstr, string $errfile, int $errline) {
        throw new ErrorException($errstr, 0, $errno, $errfile, $errline);
    }

    public static function handleException(Throwable $e) {
        // définition du code d'erreur
        http_response_code(500);
        // attention, en cas d'environnement de production, on ne voudrait pas montrer toutes ces infos
        echo json_encode([
          "code" => $e->getCode(),
          "message" => $e->getMessage(),
          "file" => $e->getFile(),
          "line" => $e->getLine()
        ]);
```

Refactorisation

En utilisant les fonctions natives de PHP `set_error_handler` et `set_exception_handler` nous allons pouvoir rediriger les erreurs et exceptions vers notre classe `ErrorHandler`.

Modifiez le début du fichier php `index.php` ainsi :

```php
ini_set("display_errors", "On");

require_once dirname(__DIR__) . "/vendor/autoload.php";

set_exception_handler([ErrorHandler::class, "handleException"]);
set_error_handler([ErrorHandler::class, "handleError"]);
... // reste du fichier
```

Implémentation GET /users

Revenons dans notre class UserController. Maintenant, nous pouvons utiliser la méthode `getAll` de notre `UserGateway` pour récupérer tous les utilisateurs.

```
switch ($method) {
    case 'GET':
        echo json_encode($this->userGateway->getAll()); // cela renvoie le tableau des utilisateurs
        break;
        ...
}
```

Si on teste, cela devrait fonctionner :

Implémentation GET /users

Ensuite, pour GET, il nous reste à implémenter la méthode `getById` pour récupérer un utilisateur par son id.

Dans la classe `UserGateway` ajoutez la méthode `getById` :

```php
public function getById(string $id)  {
    $statement = $this->connection->prepare("SELECT * FROM users WHERE id = :id LIMIT 1");
    $statement->bindValue(":id", $id, PDO::PARAM_INT);

    $statement->execute();
    $result = $statement->fetch(PDO::FETCH_ASSOC);
    if ($result) {
        return  [
            "id" => $result["id"],
            "first_name" => $result["first_name"],
            "last_name" => $result["first_name"],
            "email" => $result["email"],
            "avatar" => $result["avatar"]
        ];
    }
    return false;
}
```

Implémentation GET /users

Et ainsi on utilise cette méthode dans le UserController:

```
switch ($method) {
    case 'GET':
        if ($id) {
            $user = $this->userGateway->getById($id);
            if ($user) {
                echo json_encode($user);
            } else {
                http_response_code(404);
                echo json_encode(["message" => "User not found"]);
            }
        } else {
            echo json_encode($this->userGateway->getAll());
        }
        break;
        ...
}
```

Implémentation POST /users

Pour la création d'un utilisateur, on suit une logique similaire. La seule différence étant que :

- on doit obtenir les données de la requête POST (qui sont dans le corps de la requête). Pour cela on utilisera `file_get_contents('php://input')` qui permet de récupérer le corps de la requête.
- on doit valider les données à minima.
- on doit appeler la méthode `insert` de notre `UserGateway`

Implémentation POST /users

Voici donc comment on peut procéder: Dans notre `UserController` , on récupère les données, on les valide et on les envoie à notre `UserGateway` :

```
...
switch ($method) {
  ...
    case 'POST':
        $data =  json_decode(file_get_contents("php://input"), true);
        // on s'assuree que les données sont valides
        $errors = $this->getValidationErrors($data);
        if (!empty($errors)) {
            $this->responseValidationError($errors);
            return;
        }
        $createdId = $this->userGateway->insert($data);

        break;
...
}
```

Implémentation POST /users

Dans cette même classe la validation des erreurs nous permets de vérifier que les données sont bien présentes et qu'elles sont valides.

```php
private function getValidationErrors(array $data): array {
   $errors = [];
    if (!isset($data["first_name"]) || empty($data["first_name"])) {
        $errors["first_name"] = "Le prénom est obligatoire";
    }
    if (!isset($data["last_name"]) || empty($data["last_name"])) {
        $errors["last_name"] = "Le nom est obligatoire";
    }
    if (!isset($data["email"]) || empty($data["email"])) {
        $errors["email"] = "L'email est obligatoire";
    }
    return $errors;
}
```

Implémentation POST /users

Enfin, dans le gateway on implémente la méthode `insert` :

```
    public function insert(array $data): string {
        $statement = $this->connection->prepare("INSERT INTO users (first_name, last_name, email, avatar) VALUES (:first
        $statement->bindValue(":first_name", $data["first_name"], PDO::PARAM_STR);
        $statement->bindValue(":last_name", $data["last_name"], PDO::PARAM_STR);
        $statement->bindValue(":email", $data["email"], PDO::PARAM_STR);
        $statement->execute();
        return $this->connection->lastInsertId(); // we want to return the id of the new user
    }
```

Implémentation POST /users

Attention, si vous avez une erreur de type `SQLSTATE[HY000]: General error: 8 attempt to write a readonly database`, c'est probablement parce à cause des droits d'écriture dans le dossier contenant la base de données. Pour fonctionner sqlite à besoin de pouvoir écrire dans le dossier contenant la base de données.

Il suffit donc de changer les droits du dossier contenant la base de données pour que le serveur web puisse écrire dedans.


```
{
    "code": "HY000",
    "file": "/var/www/html/src/UserGateway.php",
    "line": 52,
    "message": "SQLSTATE[HY000]: General error: 8 attempt to write a readonly database"
}
```

Ainsi désormais, on devrait avoir nos méthodes GET et POST qui fonctionne

Création d'un User avec httpie (ou postman) :

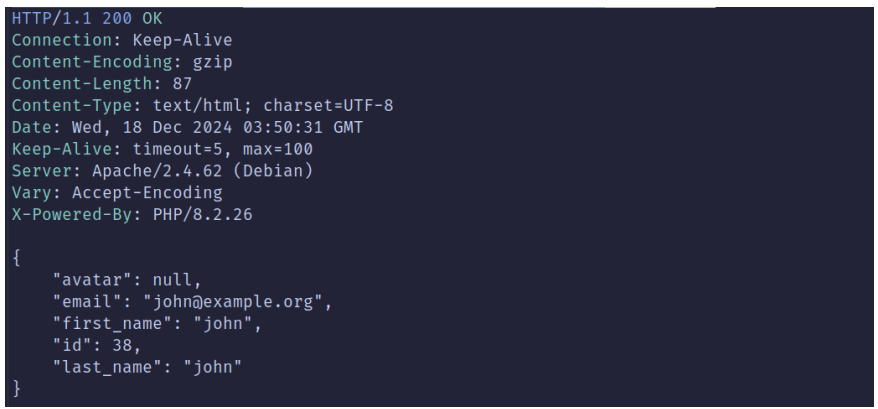
```
http post http://localhost:8080/api/users first_name=John last_name=Doe email=john@example.com
```

```
HTTP/1.1 201 Created
Connection: Keep-Alive
Content-Length: 36
Content-Type: text/html; charset=UTF-8
Date: Wed, 18 Dec 2024 03:49:50 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.62 (Debian)
X-Powered-By: PHP/8.2.26

{
    "id": "38",
    "message": "User Created"
}
```

Ainsi désormais, on devrait avoir nos méthodes GET et POST qui fonctionne

On voit également que l'on peut récupérer un utilisateur par son id :

```
http get http://localhost:8080/api/users/38
```



Conclusion

- Nous avons vu comment créer une API Rest basique en PHP.
- Nous avons un système de routage qui nous permet de gérer les différentes routes de notre API.
- Nous avons un système de gestion des erreurs qui nous permet de renvoyer des messages d'erreur en cas de problème.
- Nous avons un système de connection à une base de données SQLite.
- Nous avons un système de gestion des requêtes CRUD qui ne gère pour l'instant que GET et POST, mais qui pourrait être facilement extensible.

Conseils

Ce TD est une base pour comprendre comment fonctionne une API Rest. Il est très basique et ne prend pas en compte de nombreux aspects comme la sécurité, la validation des données, la pagination, la gestion des erreurs, etc.

Néanmoins le patron de conception est relativement fidèle à ce que l'on peut retrouver dans une API Rest classique.

Familiarisez-vous avec les concepts de base, et n'hésitez pas à explorer d'autres aspects de la création d'une API Rest.