API – REST FULL

**ESTRUCTURA DE UNA API EN PHP CON POO**

Desarrollar una API en PHP utilizando Programación Orientada a Objetos (POO) puede ser muy eficiente y escalable. A continuación, te proporcionaré una estructura básica de cómo podrías organizar tu código:

**1. Configuración Inicial**

Empieza creando un archivo de configuración para la base de datos (config.php):

php

<?php

class Database {

private $host = "localhost";

private $db\_name = "tu\_base\_de\_datos";

private $username = "tu\_usuario";

private $password = "tu\_contraseña";

public $conn;

public function getConnection() {

$this->conn = null;

try {

$this->conn = new PDO("mysql:host=" . $this->host . ";dbname=" . $this->db\_name, $this->username, $this->password);

$this->conn->exec("set names utf8");

} catch(PDOException $exception) {

echo "Error de conexión: " . $exception->getMessage();

}

return $this->conn;

}

}

?>

**2. Crear Clases para los Modelos**

Crea una clase para manejar las operaciones relacionadas con tu entidad. Por ejemplo, para los usuarios (User.php):

php

<?php

class User {

private $conn;

private $table\_name = "users";

public $id;

public $name;

public $email;

public function \_\_construct($db) {

$this->conn = $db;

}

// Método para leer todos los usuarios

public function read() {

$query = "SELECT id, name, email FROM " . $this->table\_name;

$stmt = $this->conn->prepare($query);

$stmt->execute();

return $stmt;

}

// Otros métodos para crear, actualizar y eliminar usuarios

}

?>

**3. Crear los Endpoints de la API**

Crea archivos para manejar las solicitudes a tus diferentes endpoints. Por ejemplo, para obtener todos los usuarios (read\_users.php):

php

<?php

header("Access-Control-Allow-Origin: \*");

header("Content-Type: application/json; charset=UTF-8");

include\_once 'config.php';

include\_once 'User.php';

$database = new Database();

$db = $database->getConnection();

$user = new User($db);

$stmt = $user->read();

$num = $stmt->rowCount();

if($num>0){

$users\_arr = array();

$users\_arr["records"] = array();

while ($row = $stmt->fetch(PDO::FETCH\_ASSOC)){

extract($row);

$user\_item = array(

"id" => $id,

"name" => $name,

"email" => $email

);

array\_push($users\_arr["records"], $user\_item);

}

echo json\_encode($users\_arr);

} else {

echo json\_encode(

array("message" => "No se encontraron usuarios.")

);

}

?>

**4.** [**Index.php**](https://index.php/?form=MG0AV3) **para Redirigir Peticiones**

Finalmente, puedes tener un archivo index.php que redirija las peticiones a los diferentes scripts:

php

<?php

$request = $\_SERVER['REQUEST\_URI'];

switch ($request) {

case '/api/read\_users':

require \_\_DIR\_\_ . '/read\_users.php';

break;

// Añadir otros casos para diferentes endpoints

default:

http\_response\_code(404);

echo json\_encode(array("message" => "Endpoint no encontrado."));

break;

}

?>

**Ventajas de usar POO:**

* **Modularidad**: Fácil de mantener y escalar.
* **Reutilización de Código**: Clases y métodos pueden ser reutilizados en diferentes partes del proyecto.
* **Encapsulamiento**: Protección de datos y lógica interna.

**FUNSIONES DE .HTACCES**

El archivo .htaccess es un archivo de configuración utilizado por servidores web como Apache. Aunque no está directamente relacionado con PHP, puede ser muy útil para configurar y optimizar cómo se maneja tu sitio web o aplicación PHP. Aquí te dejo algunas de sus funciones principales:

**1. Reescritura de URL (URL Rewrite)**

Permite crear URLs amigables o redirigir peticiones. Por ejemplo, convertir example.com/index.php?page=about en example.com/about.

plaintext

RewriteEngine On

RewriteRule ^about$ index.php?page=about [L]

**2. Redirección (Redirect)**

Redirige a los usuarios de una URL antigua a una nueva.

plaintext

Redirect 301 /old-page.html <http://www.example.com/new-page.html>

**3. Protección de Directorios y Archivos**

Puedes restringir el acceso a ciertos archivos o directorios.

plaintext

<Files "config.php">

Order Allow,Deny

Deny from all

</Files>

**4. Control de Caché**

Configura reglas de caché para mejorar el rendimiento del sitio.

plaintext

<FilesMatch "\.(jpg|png|gif|js|css)$">

Header set Cache-Control "max-age=31536000, public"

</FilesMatch>

**5. Configuración de Errores Personalizados**

Permite definir páginas de error personalizadas.

plaintext

ErrorDocument 404 /404.html

ErrorDocument 500 /500.html

**6. Autenticación y Autorización**

Puedes proteger directorios con contraseña.

plaintext

AuthType Basic

AuthName "Restricted Content"

AuthUserFile /path/to/.htpasswd

Require valid-user

**7. Control de Acceso por IP**

Permite limitar el acceso a tu sitio web según la dirección IP.

plaintext

Order Deny,Allow

Deny from all

Allow from 192.168.1.100

**8. Desactivar Listado de Directorios**

Evita que los visitantes vean un listado de archivos en directorios sin un archivo de índice.

plaintext

Options -Indexes

Estas son solo algunas de las muchas funciones que puedes configurar con un archivo .htaccess. Este archivo es muy potente y permite una gran personalización del comportamiento del servidor web.

**DATOS JSON EN PHP POO**

Trabajar con datos JSON en PHP utilizando Programación Orientada a Objetos (POO) es muy útil para organizar tu código y hacerlo más modular y mantenible. Aquí tienes una guía sobre cómo hacerlo.

**Paso 1: Crear la Configuración de la Base de Datos**

Primero, definimos una clase para manejar la conexión a la base de datos. Esto nos permitirá reutilizar esta conexión en otros lugares de nuestro código.

php

<?php

class Database {

private $host = "localhost";

private $db\_name = "tu\_base\_de\_datos";

private $username = "tu\_usuario";

private $password = "tu\_contraseña";

public $conn;

public function getConnection() {

$this->conn = null;

try {

$this->conn = new PDO("mysql:host=" . $this->host . ";dbname=" . $this->db\_name, $this->username, $this->password);

$this->conn->exec("set names utf8");

} catch(PDOException $exception) {

echo "Error de conexión: " . $exception->getMessage();

}

return $this->conn;

}

}

?>

**Paso 2: Crear la Clase del Modelo**

Definimos una clase para representar la entidad que manejaremos. En este caso, usaremos una clase User para los usuarios.

php

<?php

class User {

private $conn;

private $table\_name = "users";

public $id;

public $name;

public $email;

public function \_\_construct($db) {

$this->conn = $db;

}

// Método para obtener todos los usuarios

public function read() {

$query = "SELECT id, name, email FROM " . $this->table\_name;

$stmt = $this->conn->prepare($query);

$stmt->execute();

return $stmt;

}

// Otros métodos para crear, actualizar y eliminar usuarios

}

?>

**Paso 3: Crear un Endpoint para Obtener Datos JSON**

Ahora, creamos un archivo PHP que maneje la solicitud para obtener datos JSON. Este archivo utilizará la clase User para obtener los datos y los devolverá en formato JSON.

php

<?php

header("Access-Control-Allow-Origin: \*");

header("Content-Type: application/json; charset=UTF-8");

include\_once 'config.php';

include\_once 'User.php';

$database = new Database();

$db = $database->getConnection();

$user = new User($db);

$stmt = $user->read();

$num = $stmt->rowCount();

if($num>0){

$users\_arr = array();

$users\_arr["records"] = array();

while ($row = $stmt->fetch(PDO::FETCH\_ASSOC)){

extract($row);

$user\_item = array(

"id" => $id,

"name" => $name,

"email" => $email

);

array\_push($users\_arr["records"], $user\_item);

}

echo json\_encode($users\_arr);

} else {

echo json\_encode(

array("message" => "No se encontraron usuarios.")

);

}

?>

**Paso 4: Crear un Archivo de Rutas (Opcional)**

Para manejar múltiples **endpoints** de una manera organizada, podemos utilizar un archivo de rutas.

php

<?php

$request = $\_SERVER['REQUEST\_URI'];

switch ($request) {

case '/api/read\_users':

require \_\_DIR\_\_ . '/read\_users.php';

break;

// Añadir otros casos para diferentes endpoints

default:

http\_response\_code(404);

echo json\_encode(array("message" => "Endpoint no encontrado."));

break;

}

?>

**Beneficios de Usar POO con JSON en PHP**

1. **Modularidad**: Puedes reutilizar las clases y métodos en diferentes partes de tu aplicación.
2. **Mantenibilidad**: El código es más fácil de mantener y escalar.
3. **Claridad**: La estructura clara del código facilita la comprensión y el desarrollo colaborativo.