Análisis de un backend PHP sin framework (Proyecto GTask)

Módulo: Desarrollo Web en Entorno Servidor (DWES)
Alumno: Francisco Alba Muñoz

Parte A — Mapa de arquitectura

Flujo general de una petición HTTP

El flujo de una petición HTTP en el proyecto GTask sigue el siguiente esquema desde que el cliente realiza una solicitud hasta que recibe una respuesta JSON:

```
Cliente HTTP → index.php (Router) → Support.php (CORS + Validación Auth)
→ Controller (AuthController/TaskController) → Database (PDO)
→ PostgreSQL → Controller → Support.php (JSON Response) → Cliente HTTP
```

Descripción detallada del flujo

1. **Recepción de la petición**: El servidor web (nginx) redirige todas las peticiones al fichero `app/public/index.php`, que actúa como punto de entrada único (front controller).
2. **Inicialización de sesión**: Se inicia la sesión PHP mediante `session_start()` para mantener el estado de autenticación del usuario entre peticiones.
3. **Carga de dependencias**: Se incluyen los ficheros necesarios:
   - `Support.php`: Funciones auxiliares para respuestas JSON, lectura de peticiones y autenticación
   - `Database.php`: Clase de conexión a base de datos
   - Controladores: `AuthController.php` y `TaskController.php`
4. **Configuración CORS**: Se ejecuta `apply_cors()` definida en `Support.php` para establecer las cabeceras de Cross-Origin Resource Sharing, permitiendo peticiones desde otros orígenes. Si la petición es de tipo OPTIONS (preflight), se responde con código 204 y se termina la ejecución.
5. **Conexión a base de datos**: Se carga el fichero de configuración `config/config.php` que retorna un array con los parámetros de conexión obtenidos de variables de entorno. Se instancia la clase `Database` pasando esta configuración.
6. **Instanciación de controladores**: Se crean instancias de `AuthController` y `TaskController`, inyectando el objeto PDO obtenido de la base de datos.
7. **Análisis de la petición**: Se obtiene el método HTTP (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`) y la ruta solicitada mediante `$_SERVER['REQUEST_METHOD']` y `parse_url()` respectivamente.
8. **Enrutamiento manual**: Se divide la ruta en segmentos usando `explode()`. La primera parte debe ser `api`, la segunda indica el recurso (`register`, `login`, `logout`, `me`, `tasks`), y la tercera (opcional) contiene el ID del recurso.
9. **Ejecución del controlador**: Según el recurso y método HTTP, se ejecuta el método correspondiente del controlador:
   - Para rutas de autenticación (`register`, `login`, `logout`, `me`): se invoca `AuthController`
   - Para rutas de tareas (`tasks`): se verifica autenticación con `require_auth()` y se invoca `TaskController`
10. **Verificación de autenticación**: Para endpoints protegidos, la función `require_auth()` valida que exista `$_SESSION['user']`. Si no existe, se devuelve un error 401 y se termina la ejecución.
11. **Procesamiento en el controlador**: El controlador:
    - Extrae los datos del cuerpo JSON usando `get_json_body()`
    - Valida los datos recibidos
    - Ejecuta consultas SQL preparadas contra PostgreSQL usando PDO
    - Retorna respuesta mediante `json_response()` o `json_error()`
12. **Respuesta JSON**: Las funciones `json_response()` y `json_error()` establecen el código HTTP, la cabecera `Content-Type: application/json`, codifican el array PHP a JSON y terminan la ejecución.

Ficheros involucrados en el flujo

- **Enrutamiento**: `app/public/index.php` (líneas 21-92)
- **Controladores**: `app/src/Controllers/AuthController.php` y `app/src/Controllers/TaskController.php`
- **Acceso a base de datos**: `app/src/Database.php` mediante PDO
- **Formato de respuesta**: Funciones `json_response()` y `json_error()` en `app/src/Support.php`

Parte B — Conexión a base de datos

Ubicación de la creación de la conexión

La conexión a la base de datos se crea en el fichero `app/src/Database.php` mediante el constructor de la clase `Database` (líneas 8-28).

La instanciación de esta clase se realiza en `app/public/index.php` (línea 18):

```
$database = new Database($config['db']);
```

Construcción del DSN (Data Source Name)

El DSN se construye utilizando `sprintf()` en el constructor de la clase `Database` (líneas 11-15):

```
$dsn = sprintf(
    'pgsql:host=%s;port=%s;dbname=%s',
    $config['host'],
    $config['port'],
    $config['name']
);
```

El DSN resultante tiene el formato: `pgsql:host=db;port=5432;dbname=app`

Obtención de los valores de configuración

Los valores de configuración se obtienen desde el fichero `app/config/config.php` (líneas 3-12), el cual retorna un array asociativo.

Cada valor se extrae de variables de entorno mediante la función `getenv()`, con un valor por defecto mediante el operador ternario `?:`:

```
'db' => [
    'host' => getenv('DB_HOST') ?: 'db',
    'port' => getenv('DB_PORT') ?: '5432',
    'name' => getenv('DB_NAME') ?: 'app',
    'user' => getenv('DB_USER') ?: 'user',
    'password' => getenv('DB_PASSWORD') ?: 'password',
]
```

Las variables de entorno permiten configurar la aplicación sin modificar el código, siguiendo el principio de configuración externalizada. Los valores por defecto garantizan que la aplicación funcione en un entorno de desarrollo local.

Configuración de PDO

El objeto PDO se crea en `app/src/Database.php` (líneas 18-27) con las siguientes opciones:

- `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION`: Configura PDO para que lance excepciones ante errores SQL, permitiendo un mejor control de errores mediante bloques try-catch.
- `PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC`: Establece que por defecto los resultados de consultas se retornen como arrays asociativos (clave-valor) en lugar de objetos o arrays numéricos.

Comportamiento ante fallo de conexión

Si la conexión a la base de datos falla, PDO lanza una excepción de tipo `PDOException` debido a la configuración `ERRMODE_EXCEPTION`.

En el código actual, esta excepción no está capturada, por lo que:

1. PHP mostrará un error fatal y detendrá la ejecución
2. Se expondrá información sensible del error (credenciales, host, etc.) al cliente
3. El usuario recibirá un error HTTP 500 (Internal Server Error)

Este comportamiento es inadecuado para producción, ya que expone detalles internos de la aplicación. Una implementación más robusta debería capturar la excepción y retornar un error genérico mediante `json_error()`.

Acceso al objeto PDO desde controladores

El método público `pdo()` en la clase `Database` (líneas 30-34) retorna la instancia de PDO creada en el constructor. Este objeto se inyecta en los controladores durante su instanciación:

```
$authController = new AuthController($database->pdo());
$taskController = new TaskController($database->pdo());
```

Ambos controladores almacenan la referencia en una propiedad privada `$pdo` para realizar consultas SQL.

Parte C — Enrutado y gestión de peticiones

Distinción del método HTTP

El método HTTP se obtiene mediante la superglobal `$_SERVER['REQUEST_METHOD']` en `app/public/index.php` (línea 22):

```
$method = $_SERVER['REQUEST_METHOD'] ?? 'GET';
```

El operador null coalescing `??` proporciona `'GET'` como valor por defecto si la clave no existe. Esta variable se utiliza posteriormente en las condiciones de enrutamiento para determinar qué acción ejecutar.

Procesamiento de la ruta

La ruta solicitada se extrae en `app/public/index.php` (líneas 23-28):

```
$path = parse_url($_SERVER['REQUEST_URI'] ?? '/', PHP_URL_PATH);
$path = rtrim($path, '/');

if ($path === '') {
    $path = '/';
}
```

Proceso:

1. `parse_url()` extrae únicamente la parte del path, descartando query strings y fragmentos
2. `rtrim()` elimina barras finales para normalizar la ruta
3. Se asegura que una ruta vacía se convierta en `/`

La ruta se divide en segmentos (línea 38):

```
$segments = explode('/', trim($path, '/'));
```

Los segmentos se utilizan para identificar:

- `$segments[0]` : Debe ser `'api'` (líneas 40-42)
- `$segments[1]` : Recurso solicitado (`$resource`) - línea 45
- `$segments[2]` : ID del recurso (opcional) - línea 46

Decisión de controlador y método a ejecutar

La decisión se toma mediante una serie de condicionales que evalúan el recurso y el método HTTP (líneas 48-89):

Para rutas de autenticación (no requieren autenticación previa):

- `/api/register` + POST → `AuthController::register()`
- `/api/login` + POST → `AuthController::login()`
- `/api/logout` + POST → `AuthController::logout()`
- `/api/me` + GET → `AuthController::me()`

Para rutas de tareas (requieren autenticación): Se ejecuta primero `require_auth()` para obtener el usuario autenticado (línea 62), y luego:

- `/api/tasks` + GET (sin ID) → `TaskController::index()`
- `/api/tasks` + POST (sin ID) → `TaskController::create()`
- `/api/tasks/{id}` + GET → `TaskController::show()`
- `/api/tasks/{id}` + PUT/PATCH → `TaskController::update()`
- `/api/tasks/{id}` + DELETE → `TaskController::delete()`

Si ninguna condición se cumple, se ejecuta `json_error('Ruta no encontrada.', 404)` al final del fichero (línea 91).

Tabla de endpoints

Endpoint	Método	Autenticación	Controller::método	Descripción
`/api/register`	POST	No	AuthController::register	Registrar nuevo usuario
`/api/login`	POST	No	AuthController::login	Iniciar sesión
`/api/logout`	POST	No	AuthController::logout	Cerrar sesión
`/api/me`	GET	Sí	AuthController::me	Obtener datos del usuario autenticado

Endpoint	Método	Autenticación	Controller::método	Descripción
`/api/tasks`	GET	Sí	TaskController::index	Listar todas las tareas del usuario
`/api/tasks`	POST	Sí	TaskController::create	Crear nueva tarea
`/api/tasks/{id}`	GET	Sí	TaskController::show	Obtener una tarea específica
`/api/tasks/{id}`	PUT	Sí	TaskController::update	Actualizar una tarea completa
`/api/tasks/{id}`	PATCH	Sí	TaskController::update	Actualizar campos parciales de una tarea
`/api/tasks/{id}`	DELETE	Sí	TaskController::delete	Eliminar una tarea

Características del enrutamiento

Ventajas:

- Simple y fácil de entender para proyectos pequeños
- No requiere dependencias externas
- Control total sobre la lógica de enrutamiento

Limitaciones:

- No es escalable para aplicaciones grandes (muchas rutas generan código extenso)
- No soporta patrones complejos de rutas (ej: expresiones regulares)
- No permite agrupación de rutas ni middlewares reutilizables
- Cada nueva ruta requiere modificar el fichero central

Parte D — Validación y control de tipos

Validaciones en el registro de usuarios

Ubicación: `app/src/Controllers/AuthController.php`, método `register()` (líneas 14-32)

Campos obligatorios (líneas 20-22):

```
if ($name === '' || $email === '' || $password === '') {
    json_error('Nombre, email y contraseña son obligatorios.', 422);
}
```

Longitud del nombre (líneas 23-25):

```
if (mb_strlen($name) > 100) {
    json_error('El nombre no puede superar 100 caracteres.', 422);
}
```

Formato de email válido (líneas 26-28):

```
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    json_error('El email no es valido.', 422);
}
```

Longitud mínima de contraseña (líneas 29-31):

```
if (mb_strlen($password) < 6) {
    json_error('La contraseña debe tener al menos 6 caracteres.', 422);
}
```

Unicidad del email (líneas 34-38):

```
$stmt = $this->pdo->prepare('SELECT id FROM users WHERE email = :email');
$stmt->execute(['email' => $email]);
if ($stmt->fetch()) {
    json_error('El email ya está registrado.', 409);
}
```

Validaciones en el login

Ubicación: `app/src/Controllers/AuthController.php`, método `login()` (líneas 64-73)

Campos obligatorios (líneas 67-69):

```
if ($email === '' || $password === '') {
    json_error('Email y contraseña son obligatorios.', 422);
}
```

Validación de formato de email (líneas 70-72):

```
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    json_error('El email no es valido.', 422);
}
```

Validación de credenciales (líneas 79-81):

```
if (!$user || !password_verify($password, $user['password'])) {
    json_error('Credenciales inválidas.', 401);
}
```

Validaciones en la creación de tareas

Ubicación: `app/src/Controllers/TaskController.php`, método `create()` (líneas 42-69)

Título obligatorio (líneas 52-54):

```
if ($title === '') {
    json_error('El título es obligatorio.', 422);
}
```

Longitud máxima del título (líneas 55-57):

```
if (mb_strlen($title) > 200) {
    json_error('El título no puede superar 200 caracteres.', 422);
}
```

Longitud máxima de la descripción (líneas 58-60):

```
if ($description !== null && mb_strlen((string)$description) > 1000) {
    json_error('La descripción no puede superar 1000 caracteres.', 422);
}
```

Valores permitidos para status (líneas 61-63):

```
if (!in_array($status, ['pending', 'completed'], true)) {
    json_error('El estado no es valido.', 422);
}
```

Rango de prioridad (líneas 64-66):

```
if ($priority < 0 || $priority > 5) {
    json_error('La prioridad debe estar entre 0 y 5.', 422);
}
```

Formato de fecha (líneas 67-69):

```
if ($dueDate !== null && !$this->isValidDate($dueDate)) {
    json_error('La fecha limite debe tener formato YYYY-MM-DD.', 422);
}
```

El método `isValidDate()` (líneas 188-191) utiliza `DateTimeImmutable::createFromFormat()` para validar estrictamente el formato YYYY-MM-DD:

```
private function isValidDate(string $date): bool
{
    $dt = DateTimeImmutable::createFromFormat('Y-m-d', $date);
    return $dt !== false && $dt->format('Y-m-d') === $date;
}
```

Validaciones en la actualización de tareas

Ubicación: `app/src/Controllers/TaskController.php`, método `update()` (líneas 97-165)

Este método implementa validaciones dinámicas según los campos enviados:

Validación de título (líneas 103-112): Si se envía el campo `title`, se valida que no esté vacío y no supere 200 caracteres.

Validación de descripción (líneas 114-120): Si se envía, se valida que no supere 1000 caracteres.

Validación de status (líneas 122-134): Si se envía, debe ser `'pending'` o `'completed'`. Si es `'completed'`, se establece automáticamente `completed_at` con la fecha y hora actual.

Validación de due_date (líneas 136-142): Si se envía, debe tener formato YYYY-MM-DD válido.

Validación de prioridad (líneas 144-150): Si se envía, debe estar entre 0 y 5.

Validación de campos a actualizar (líneas 152-154): Se verifica que al menos un campo haya sido enviado para actualizar.

Control de tipos

El proyecto utiliza PHP 7.4+ con declaración estricta de tipos (`declare(strict_types=1);`) en `app/public/index.php` (línea 3).

Type hints en parámetros y retornos:

- `public function register(array $data): void` (AuthController, línea 14)
- `public function __construct(PDO $pdo)` (ambos controladores)
- `private function isValidDate(string $date): bool` (TaskController, línea 188)

Conversiones explícitas de tipo:

- `$userId = (int)$user['id'];` (index.php, línea 64)
- `$taskId = (int)$stmt->fetchColumn();` (TaskController, línea 93)
- `$priority = (int)$data['priority'];` (TaskController, línea 145)
- `$email = strtolower(trim($data['email'] ?? ''));` (AuthController, líneas 17, 66)

Uso de mb_strlen(): Se utiliza `mb_strlen()` en lugar de `strlen()` para soportar correctamente caracteres multibyte (UTF-8), garantizando que la validación de longitud sea precisa independientemente del idioma.

Códigos de error HTTP utilizados

Código	Significado	Uso en la aplicación
200	OK	Respuesta exitosa por defecto (json_response)
201	Created	Usuario registrado exitosamente
204	No Content	Respuesta a peticiones OPTIONS (CORS preflight)
400	Bad Request	JSON inválido en el cuerpo de la petición
401	Unauthorized	Usuario no autenticado o credenciales inválidas
404	Not Found	Ruta, tarea o recurso no encontrado
409	Conflict	Email ya registrado (duplicado)
422	Unprocessable Entity	Errores de validación de datos

Ejemplos de datos inválidos y respuestas esperadas

Ejemplo 1: Registro con email duplicado

```
Request POST /api/register
{
  "name": "Juan Pérez",
  "email": "juan@example.com",
  "password": "123456"
}

Response 409 Conflict
{
  "error": "El email ya está registrado."
}
```

Ejemplo 2: Creación de tarea con título vacío

```
Request POST /api/tasks
{
  "title": "",
  "description": "Descripción de prueba"
}

Response 422 Unprocessable Entity
{
  "error": "El título es obligatorio."
}
```

Ejemplo 3: Actualización de tarea con prioridad fuera de rango

```
Request PUT /api/tasks/5
{
  "priority": 10
}

Response 422 Unprocessable Entity
{
  "error": "La prioridad debe estar entre 0 y 5."
}
```

Ejemplo 4: Acceso sin autenticación

```
Request GET /api/tasks

Response 401 Unauthorized
{
  "error": "No autenticado."
}
```

Ejemplo 5: Fecha con formato incorrecto

```
Request POST /api/tasks
{
  "title": "Tarea urgente",
  "due_date": "2025/01/30"
}

Response 422 Unprocessable Entity
{
  "error": "La fecha limite debe tener formato YYYY-MM-DD."
}
```

Parte E — Autenticación y control de acceso

Gestión de la sesión del usuario

La sesión se gestiona utilizando el mecanismo de sesiones nativo de PHP mediante `$_SESSION`.

Inicio de sesión (app/public/index.php, línea 6):

```
session_start();
```

Esta función debe ejecutarse antes de cualquier salida al navegador. Crea o reanuda una sesión identificada por una cookie (PHPSESSID por defecto) que se envía al cliente.

Almacenamiento en sesión tras registro exitoso (app/src/Controllers/AuthController.php, líneas 54-58):

```
$_SESSION['user'] = [
    'id' => $userId,
    'name' => $name,
    'email' => $email,
];
```

Almacenamiento en sesión tras login exitoso (app/src/Controllers/AuthController.php, líneas 85-89):

```
$_SESSION['user'] = [
    'id' => (int)$user['id'],
    'name' => $user['name'],
    'email' => $user['email'],
];
```

Destrucción de sesión en logout (app/src/Controllers/AuthController.php, líneas 97-99):

```
$_SESSION = [];
session_destroy();
json_response(['message' => 'Sesión cerrada.']);
```

Información guardada en $_SESSION

La clave `$_SESSION['user']` contiene un array asociativo con tres campos:

- **id**: Identificador único del usuario en la base de datos (integer)
- **name**: Nombre completo del usuario (string)
- **email**: Dirección de correo electrónico (string, normalizado a minúsculas)

Esta información se almacena en el servidor. El cliente únicamente recibe una cookie con el identificador de sesión, lo que previene la manipulación de datos por parte del usuario.

Bloqueo de acceso a endpoints protegidos

El bloqueo se realiza mediante la función `require_auth()` definida en `app/src/Support.php` (líneas 78-85):

```
function require_auth(): array
{
    // La sesion guarda el usuario autenticado.
    if (empty($_SESSION['user'])) {
        json_error('No autenticado.', 401);
    }

    return $_SESSION['user'];
}
```

Funcionamiento:

1. Verifica que `$_SESSION['user']` exista y no esté vacío
2. Si no existe, llama a `json_error()` que establece código HTTP 401, retorna JSON de error y termina la ejecución con `exit`
3. Si existe, retorna el array del usuario para uso posterior

Uso en endpoints protegidos (app/public/index.php, líneas 62-64):

```
if ($resource === 'tasks') {
    $user = require_auth();
    $userId = (int)$user['id'];
    // ... resto de lógica de tareas
}
```

También se utiliza en `AuthController::me()` (línea 106):

```
public function me(): void
{
    $user = require_auth();
    json_response(['user' => $user]);
}
```

Prevención de acceso a recursos de otros usuarios

El control de acceso se implementa mediante la cláusula `user_id` en las consultas SQL, garantizando que cada usuario solo pueda acceder a sus propios recursos.

Listado de tareas (app/src/Controllers/TaskController.php, líneas 15-18):

```
$stmt = $this->pdo->prepare(
    'SELECT ... FROM tasks WHERE user_id = :user_id ORDER BY created_at DESC'
);
$stmt->execute(['user_id' => $userId]);
```

Obtención de una tarea (app/src/Controllers/TaskController.php, líneas 26-29):

```
$stmt = $this->pdo->prepare(
    'SELECT ... FROM tasks WHERE id = :id AND user_id = :user_id'
);
$stmt->execute(['id' => $taskId, 'user_id' => $userId]);
```

Actualización de una tarea (app/src/Controllers/TaskController.php, línea 159):

```
$sql = 'UPDATE tasks SET ... WHERE id = :id AND user_id = :user_id';
```

Eliminación de una tarea (app/src/Controllers/TaskController.php, línea 177):

```
$stmt = $this->pdo->prepare('DELETE FROM tasks WHERE id = :id AND user_id = :user_id');
```

Mecanismo de protección:

1. El `userId` se obtiene de `$_SESSION['user']['id']`, que fue establecido durante el login
2. Todas las consultas incluyen `user_id = :user_id` en la cláusula WHERE
3. Si un usuario intenta acceder a una tarea que no le pertenece (diferente user_id), la consulta no retorna resultados
4. Se devuelve un error 404 "Tarea no encontrada", sin revelar que el recurso existe pero pertenece a otro usuario

Ejemplo de intento de acceso no autorizado:

Usuario A (ID: 1) intenta acceder a la tarea con ID 50 que pertenece al Usuario B (ID: 2):

```
GET /api/tasks/50
$userId = 1 (de $_SESSION['user']['id'])
$taskId = 50

Query: SELECT ... FROM tasks WHERE id = 50 AND user_id = 1
Resultado: 0 filas (la tarea 50 tiene user_id = 2)

Response 404: {"error": "Tarea no encontrada."}
```

Ficheros donde se realizan las comprobaciones

Verificación de autenticación:

- `app/src/Support.php` : Función `require_auth()` (líneas 78-85)
- `app/public/index.php` : Invocación para recursos protegidos (líneas 62-63)
- `app/src/Controllers/AuthController.php` : Método `me()` (línea 106)

Control de acceso a recursos propios:

- `app/src/Controllers/TaskController.php` : Todos los métodos que consultan tareas
  - `index()` líneas 15-18
  - `show()` líneas 26-29
  - `update()` línea 159
  - `delete()` línea 177

Limitaciones del sistema de autenticación actual

1. **Sesiones basadas en servidor**: No es escalable horizontalmente sin configuración adicional (sesiones compartidas)
2. **Sin renovación automática**: Las sesiones no expiran automáticamente por inactividad
3. **Vulnerable a CSRF**: No implementa tokens CSRF para operaciones de modificación
4. **Sin control de concurrencia**: Múltiples sesiones activas del mismo usuario no se gestionan
5. **Dependencia de cookies**: No funciona correctamente con clientes que no soportan cookies (apps móviles nativas)

Parte F — CORS, respuestas y errores

Gestión de cabeceras CORS

Las cabeceras CORS (Cross-Origin Resource Sharing) se gestionan mediante la función `apply_cors()` definida en `app/src/Support.php` (líneas 5-43).

Parámetros de configuración CORS (líneas 8-16):

Todos los parámetros se obtienen de variables de entorno para facilitar la configuración sin modificar código:

```
$enabled = getenv('CORS_ENABLED');
$origin = $_SERVER['HTTP_ORIGIN'] ?? '';
$allowedOrigins = getenv('ALLOWED_ORIGINS') ?: '';
$allowedMethods = getenv('ALLOWED_METHODS') ?: 'GET,POST,PUT,PATCH,DELETE,OPTIONS';
$allowedHeaders = getenv('ALLOWED_HEADERS') ?: 'Content-Type';
$allowCredentials = getenv('ALLOW_CREDENTIALS') ?: 'true';
```

Desactivación de CORS (líneas 9-11): Si la variable `CORS_ENABLED` existe y su valor es `'false'`, la función retorna inmediatamente sin establecer cabeceras CORS.

Procesamiento de orígenes permitidos (líneas 18-20):

```
$origins = array_filter(array_map('trim', explode(',', $allowedOrigins)));
$allowAll = in_array('*', $origins, true);
$originAllowed = $origin !== '' && ($allowAll || in_array($origin, $origins, true));
```

Se convierte la cadena de orígenes permitidos en un array, se elimina espacios en blanco y se verifica si se permite cualquier origen (`*`) o si el origen de la petición está en la lista.

Establecimiento de Access-Control-Allow-Origin (líneas 23-28):

```
if ($originAllowed) {
    header('Access-Control-Allow-Origin: ' . $origin);
    header('Vary: Origin');
} elseif ($allowAll && strtolower($allowCredentials) !== 'true') {
    header('Access-Control-Allow-Origin: *');
}
```

Lógica aplicada:

- Si el origen está permitido específicamente, se establece `Access-Control-Allow-Origin` con ese origen exacto y se añade `Vary: Origin` para indicar que la respuesta varía según el origen
- Si se permite cualquier origen (`*`) pero NO se permiten credenciales, se establece `Access-Control-Allow-Origin: *`
- Si se permiten credenciales (`ALLOW_CREDENTIALS=true`), NO se puede usar `*` (restricción del estándar CORS)

Otras cabeceras CORS (líneas 30-35):

```
header('Access-Control-Allow-Methods: ' . $allowedMethods);
header('Access-Control-Allow-Headers: ' . $allowedHeaders);

if (strtolower($allowCredentials) === 'true') {
    header('Access-Control-Allow-Credentials: true');
}
```

- `Access-Control-Allow-Methods` : Métodos HTTP permitidos
- `Access-Control-Allow-Headers` : Cabeceras que el cliente puede enviar
- `Access-Control-Allow-Credentials` : Permite envío de cookies y credenciales

Invocación (app/public/index.php, línea 15):

```
apply_cors();
```

Se ejecuta antes del enrutamiento para que todas las respuestas incluyan las cabeceras CORS.

Respuesta a peticiones OPTIONS (preflight)

Las peticiones OPTIONS son solicitudes preflight que los navegadores envían automáticamente antes de peticiones CORS complejas (con métodos distintos a GET/POST o cabeceras personalizadas).

Manejo de OPTIONS (app/src/Support.php, líneas 38-41):

```
if (($_SERVER['REQUEST_METHOD'] ?? '') === 'OPTIONS') {
    http_response_code(204);
    exit;
}
```

Proceso:

1. Se verifica si el método HTTP es OPTIONS
2. Se establece el código de respuesta 204 (No Content)
3. Se termina la ejecución con `exit` sin devolver cuerpo en la respuesta

El código 204 indica que la petición fue exitosa pero no hay contenido que retornar. Las cabeceras CORS ya fueron establecidas anteriormente, por lo que el navegador sabe si puede proceder con la petición real.

Flujo completo de una petición CORS:

```
1. Cliente (navegador) en http://localhost:3000 quiere hacer POST /api/tasks
2. Navegador detecta que es petición CORS compleja
3. Navegador envía automáticamente OPTIONS /api/tasks
4. apply_cors() establece cabeceras CORS
5. Se detecta método OPTIONS y se responde 204
6. Navegador verifica las cabeceras recibidas
7. Si las cabeceras lo permiten, navegador envía la petición POST real
8. El servidor procesa el POST normalmente
```

Formato uniforme de respuestas JSON

Todas las respuestas de la API utilizan la función `json_response()` definida en `app/src/Support.php` (líneas 45-53):

```
function json_response(array $payload, int $status = 200): void
{
    http_response_code($status);
    header('Content-Type: application/json; charset=utf-8');
    echo json_encode($payload, JSON_UNESCAPED_UNICODE);
    exit;
}
```

Características:

1. Establece el código HTTP (parámetro `$status` , por defecto 200)
2. Establece la cabecera `Content-Type: application/json; charset=utf-8`
3. Codifica el array PHP a JSON usando `json_encode()`
4. La opción `JSON_UNESCAPED_UNICODE` evita escapar caracteres Unicode, permitiendo acentos y caracteres especiales legibles
5. Termina la ejecución con `exit` para evitar salida adicional

Ejemplos de respuestas exitosas:

Login exitoso (AuthController::login, línea 91):

```
{
  "message": "Login correcto.",
  "user": {
    "id": 1,
    "name": "Juan Pérez",
    "email": "juan@example.com"
  }
}
```

Listado de tareas (TaskController::index, línea 20):

```
{
  "tasks": [
    {
      "id": 1,
      "title": "Completar informe",
      "description": "Informe mensual de ventas",
      "status": "pending",
      "due_date": "2026-02-15",
      "priority": 3,
      "created_at": "2026-01-20 10:30:00",
      "updated_at": "2026-01-20 10:30:00",
      "completed_at": null
    }
  ]
}
```

Estructura de errores

Todos los errores utilizan la función `json_error()` definida en `app/src/Support.php` (líneas 55-59):

```
function json_error(string $message, int $status = 400, array $extra = []): void
{
    json_response(array_merge(['error' => $message], $extra), $status);
}
```

Características:

1. Siempre incluye la clave `error` con el mensaje descriptivo
2. Permite parámetros adicionales mediante el array `$extra`
3. Establece el código HTTP apropiado (parámetro `$status`, por defecto 400)
4. Internamente usa `json_response()`, garantizando formato consistente
5. Termina la ejecución, evitando procesamiento adicional

Ejemplos de respuestas de error:

Error de validación - código 422:

```
{
  "error": "El título es obligatorio."
}
```

Error de autenticación - código 401:

```
{
  "error": "No autenticado."
}
```

Recurso no encontrado - código 404:

```
{
  "error": "Tarea no encontrada."
}
```

JSON inválido - código 400:

```
{
  "error": "JSON inválido."
}
```

Conflicto - código 409:

```
{
  "error": "El email ya está registrado."
}
```

Ventajas del formato uniforme

1. **Previsibilidad**: Los clientes de la API saben exactamente qué estructura esperar
2. **Facilidad de parsing**: Siempre hay una clave `error` para errores y claves específicas para éxitos
3. **Consistencia**: Todas las respuestas son JSON, nunca HTML o texto plano
4. **Internacionalización**: El uso de `JSON_UNESCAPED_UNICODE` permite mensajes en cualquier idioma
5. **Debugging**: La estructura clara facilita la depuración en desarrollo

Manejo de errores no capturados

El sistema actual tiene limitaciones en el manejo de errores no capturados:

- **Errores de PHP**: Si ocurre un error de PHP (warning, notice, fatal error), se mezclará con la salida JSON, rompiendo el formato
- **Excepciones no capturadas**: Si PDO lanza una excepción que no se captura, PHP mostrará el error por defecto
- **Errores de conexión a BD**: No hay try-catch en la creación de la conexión PDO

Una mejora consistiría en implementar un manejador global de excepciones y errores que siempre retorne JSON.

Parte G — Propuesta de mejoras

Mejora 1: Implementación de capa de servicios para separar lógica de negocio

Problema identificado:

Actualmente, los controladores (`AuthController` y `TaskController`) contienen mezcladas tres responsabilidades:

1. Validación de datos de entrada
2. Lógica de negocio (hashing de contraseñas, cálculo de `completed_at` , etc.)
3. Acceso a base de datos (consultas SQL)

Esta mezcla de responsabilidades viola el principio de responsabilidad única (Single Responsibility Principle) y dificulta:

- La reutilización de lógica de negocio
- El testing unitario (se requiere base de datos para probar validaciones)
- El mantenimiento (cambios en la lógica de negocio requieren modificar controladores)
- La legibilidad del código (métodos extensos con múltiples niveles de abstracción)

Solución propuesta:

Crear una capa de servicios (`app/src/Services/`) que encapsule la lógica de negocio y una capa de repositorios (`app/src/Repositories/`) para el acceso a datos.

Estructura propuesta:

```
app/src/
├── Controllers/
│   ├── AuthController.php      (solo validación HTTP y orquestación)
│   └── TaskController.php      (solo validación HTTP y orquestación)
├── Services/
│   ├── AuthService.php         (lógica de autenticación)
│   └── TaskService.php         (lógica de tareas)
└── Repositories/
    ├── UserRepository.php      (acceso a datos de usuarios)
    └── TaskRepository.php      (acceso a datos de tareas)
```

Ejemplo de implementación:

TaskRepository.php:

```php
class TaskRepository
{
    private PDO $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function findByUserId(int $userId): array
    {
        $stmt = $this->pdo->prepare(
            'SELECT * FROM tasks WHERE user_id = :user_id ORDER BY created_at DESC'
        );
        $stmt->execute(['user_id' => $userId]);
        return $stmt->fetchAll();
    }

    public function findByIdAndUserId(int $id, int $userId): ?array
    {
        $stmt = $this->pdo->prepare(
            'SELECT * FROM tasks WHERE id = :id AND user_id = :user_id'
        );
        $stmt->execute(['id' => $id, 'user_id' => $userId]);
        $result = $stmt->fetch();
        return $result ?: null;
    }

    public function create(array $data): int
    {
        $stmt = $this->pdo->prepare(
            'INSERT INTO tasks (user_id, title, description, status, due_date, priority, completed_at)
             VALUES (:user_id, :title, :description, :status, :due_date, :priority, :completed_at)
             RETURNING id'
        );
        $stmt->execute($data);
        return (int)$stmt->fetchColumn();
    }

    // ... métodos update() y delete()
}
```

TaskService.php:

```
class TaskService
{
    private TaskRepository $taskRepository;

    public function __construct(TaskRepository $taskRepository)
    {
        $this->taskRepository = $taskRepository;
    }

    public function createTask(int $userId, array $data): array
    {
        $taskData = [
            'user_id' => $userId,
            'title' => trim($data['title']),
            'description' => $data['description'] ?? null,
            'status' => $data['status'] ?? 'pending',
            'due_date' => $data['due_date'] ?? null,
            'priority' => $data['priority'] ?? 0,
            'completed_at' => $this->calculateCompletedAt($data['status'] ?? 'pending'),
        ];

        $taskId = $this->taskRepository->create($taskData);
        return $this->taskRepository->findByIdAndUserId($taskId, $userId);
    }

    private function calculateCompletedAt(string $status): ?string
    {
        return $status === 'completed'
            ? (new DateTimeImmutable('now'))->format('Y-m-d H:i:s')
            : null;
    }

    // ... otros métodos de lógica de negocio
}
```

TaskController.php (refactorizado):

```
class TaskController
{
    private TaskService $taskService;

    public function __construct(TaskService $taskService)
    {
        $this->taskService = $taskService;
    }

    public function create(int $userId, array $data): void
    {
        // Validación de entrada
        $this->validateCreateData($data);

        // Delegación a la capa de servicio
        $task = $this->taskService->createTask($userId, $data);

        json_response(['task' => $task], 201);
    }

    private function validateCreateData(array $data): void
    {
        if (empty(trim($data['title'] ?? ''))) {
            json_error('El título es obligatorio.', 422);
        }
        // ... resto de validaciones
    }
}
```

Beneficios de esta mejora:

1. **Separación de responsabilidades**: Cada capa tiene una función clara y única
2. **Testabilidad**: Los servicios pueden testearse sin base de datos usando mocks de repositorios
3. **Reutilización**: La lógica de negocio puede invocarse desde múltiples controladores o incluso comandos CLI
4. **Mantenibilidad**: Cambios en la lógica de negocio no afectan al código de acceso a datos
5. **Legibilidad**: Código más limpio y expresivo con métodos pequeños y especializados
6. **Extensibilidad**: Facilita agregar nuevas funcionalidades sin modificar código existente

Esfuerzo de implementación:

- Dificultad: Media
- Tiempo estimado: 8-12 horas para refactorizar el código existente
- Requiere: Conocimientos de patrones de diseño (Repository, Service Layer)
- Retrocompatibilidad: Se puede implementar gradualmente, refactorizando un controlador a la vez

Mejora 2: Sistema de autenticación basado en tokens JWT

Problema identificado:

El sistema actual utiliza sesiones PHP (`$_SESSION`) para mantener la autenticación del usuario. Este enfoque presenta varias limitaciones:

1. **Escalabilidad horizontal limitada**: Las sesiones se almacenan en el servidor, dificultando la distribución de carga entre múltiples servidores sin configuración adicional (sesiones compartidas en Redis/Memcached)
2. **Incompatibilidad con arquitecturas modernas**: No funciona bien con:
    - Aplicaciones móviles nativas (iOS/Android)
    - Single Page Applications (SPA) con dominios separados
    - Arquitecturas de microservicios
3. **Dependencia de cookies**: Requiere que el cliente soporte y envíe cookies, lo que puede bloquearse por políticas de privacidad o configuraciones del navegador
4. **Estado en el servidor**: Requiere almacenamiento en servidor, memoria o base de datos para mantener las sesiones
5. **Sin control de expiración granular**: Las sesiones PHP expiran por inactividad, pero no hay control preciso sobre el tiempo de vida del token

Solución propuesta:

Implementar autenticación basada en JSON Web Tokens (JWT), específicamente utilizando el estándar JWS (JSON Web Signature) con algoritmo HS256 (HMAC-SHA256).

Arquitectura propuesta:

Flujo de autenticación con JWT:

```
1. Cliente envía credenciales → POST /api/login
2. Servidor valida credenciales
3. Servidor genera JWT firmado con clave secreta
4. Servidor retorna JWT al cliente
5. Cliente almacena JWT (localStorage, memoria, keychain)
6. Cliente envía JWT en cabecera Authorization: Bearer <token>
7. Servidor valida firma y claims del JWT
8. Servidor extrae user_id del payload y procesa la petición
```

Implementación sugerida:

Instalar librería JWT para PHP:

```
composer require firebase/php-jwt
```

Ejemplo de generación de token en AuthController:

```php
 use Firebase\JWT\JWT;
use Firebase\JWT\Key;

class AuthController
{
    private PDO $pdo;
    private string $jwtSecret;
    private int $jwtExpiration;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
        $this->jwtSecret = getenv('JWT_SECRET') ?: 'default-secret-change-in-production';
        $this->jwtExpiration = (int)(getenv('JWT_EXPIRATION') ?: 86400); // 24 horas
    }

    public function login(array $data): void
    {
        $email = strtolower(trim($data['email'] ?? ''));
        $password = $data['password'] ?? '';

        // Validaciones...

        $stmt = $this->pdo->prepare('SELECT id, name, email, password FROM users WHERE email = :email');
        $stmt->execute(['email' => $email]);
        $user = $stmt->fetch();

        if (!$user || !password_verify($password, $user['password'])) {
            json_error('Credenciales inválidas.', 401);
        }

        // Generar JWT
        $issuedAt = time();
        $expirationTime = $issuedAt + $this->jwtExpiration;

        $payload = [
            'iat' => $issuedAt,           // Issued at
            'exp' => $expirationTime,     // Expiration
            'iss' => 'gtask-api',         // Issuer
            'sub' => (int)$user['id'],    // Subject (user ID)
            'name' => $user['name'],
            'email' => $user['email'],
        ];

        $token = JWT::encode($payload, $this->jwtSecret, 'HS256');

        json_response([
            'message' => 'Login correcto.',
            'token' => $token,
            'expires_in' => $this->jwtExpiration,
            'user' => [
                'id' => (int)$user['id'],
                'name' => $user['name'],
                'email' => $user['email'],
            ]
        ]);
    }
}
```

Modificación de require_auth() en Support.php:

```
use Firebase\JWT\JWT;
use Firebase\JWT\Key;
use Firebase\JWT\ExpiredException;
use Firebase\JWT\SignatureInvalidException;

function require_auth(): array
{
    $authHeader = $_SERVER['HTTP_AUTHORIZATION'] ?? '';

    if (!preg_match('/Bearer\s+(.*)$/i', $authHeader, $matches)) {
        json_error('Token no proporcionado.', 401);
    }

    $token = $matches[1];
    $jwtSecret = getenv('JWT_SECRET') ?: 'default-secret-change-in-production';

    try {
        $decoded = JWT::decode($token, new Key($jwtSecret, 'HS256'));

        return [
            'id' => $decoded->sub,
            'name' => $decoded->name,
            'email' => $decoded->email,
        ];

    } catch (ExpiredException $e) {
        json_error('Token expirado.', 401);
    } catch (SignatureInvalidException $e) {
        json_error('Token inválido.', 401);
    } catch (Exception $e) {
        json_error('Error de autenticación.', 401);
    }
}
```

Configuración en docker-compose.yml:

```
environment:
  JWT_SECRET: "clave-secreta-muy-segura-256-bits-minimo"
  JWT_EXPIRATION: "3600"  # 1 hora en segundos
```

Ejemplo de petición del cliente:

```
GET /api/tasks HTTP/1.1
Host: localhost
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...
```

Beneficios de esta mejora:

1. **Stateless**: No requiere almacenamiento de sesiones en el servidor, el token contiene toda la información necesaria
2. **Escalabilidad horizontal**: Cualquier servidor puede validar el token sin compartir estado, ideal para balanceadores de carga
3. **Compatibilidad multiplataforma**: Funciona perfectamente con apps móviles, SPAs, y clientes que no soportan cookies
4. **Control de expiración preciso**: Se puede definir exactamente cuándo expira el token mediante el claim `exp`
5. **Seguridad mejorada**: La firma criptográfica previene manipulación del payload
6. **Portabilidad**: El token puede enviarse en headers, query params o body según las necesidades
7. **Interoperabilidad**: JWT es un estándar ampliamente soportado por múltiples lenguajes y plataformas
8. **Información embebida**: El payload puede contener claims adicionales (roles, permisos) sin consultar la base de datos

Consideraciones de seguridad:

1. **Clave secreta robusta**: Usar una clave de al menos 256 bits generada aleatoriamente
2. **HTTPS obligatorio**: Los tokens deben transmitirse siempre por HTTPS en producción
3. **Tiempo de expiración adecuado**: Balancear seguridad (tokens de corta duración) con UX
4. **Refresh tokens**: Implementar tokens de refresco para renovar tokens expirados sin reautenticación
5. **Lista negra de tokens**: Para logout efectivo, considerar una lista de tokens revocados
6. **Validación de claims**: Verificar siempre `iss`, `exp`, `iat` para prevenir ataques

Esfuerzo de implementación:

- Dificultad: Media-Alta
- Tiempo estimado: 6-8 horas incluyendo testing
- Requiere: Instalación de dependencia (firebase/php-jwt), modificación de AuthController y Support.php
- Retrocompatibilidad: Se puede mantener sesiones PHP temporalmente y soportar ambos métodos durante transición

Alternativa híbrida:

Mantener sesiones PHP para la aplicación web tradicional y ofrecer JWT como opción para clientes API, detectando automáticamente el método de autenticación:

```
function require_auth(): array
{
    // Intentar JWT primero
    if (!empty($_SERVER['HTTP_AUTHORIZATION'])) {
        return require_auth_jwt();
    }

    // Fallback a sesión PHP
    if (!empty($_SESSION['user'])) {
        return $_SESSION['user'];
    }

    json_error('No autenticado.', 401);
}
```

Conclusiones

El proyecto GTask constituye una implementación funcional de un backend PHP sin framework que expone de forma clara los fundamentos de las aplicaciones web en entorno servidor. El análisis realizado permite comprender los mecanismos esenciales que los frameworks modernos abstraen:

Aspectos destacados:

1. **Arquitectura clara**: El patrón front controller centraliza el enrutamiento, facilitando el mantenimiento del flujo de peticiones.
2. **Seguridad implementada**: Utiliza consultas preparadas (prevención de SQL injection), hashing de contraseñas con `password_hash()`, validación exhaustiva de entrada, y control de acceso basado en `user_id`.
3. **Separación de responsabilidades**: Aunque mejorable, existe separación entre configuración, lógica de acceso a datos, controladores y funciones auxiliares.
4. **API RESTful coherente**: Respeta convenciones REST para recursos, métodos HTTP y códigos de estado.
5. **Configuración externalizada**: Uso de variables de entorno para parámetros sensibles y de configuración.

Limitaciones identificadas:

1. **Escalabilidad**: El enrutamiento manual no escala bien con muchos endpoints.
2. **Falta de abstracción**: La lógica de negocio está mezclada con controladores y acceso a datos.
3. **Manejo de errores limitado**: No hay captura global de excepciones ni logging estructurado.
4. **Sin testing**: No existe suite de tests automatizados.
5. **Autenticación acoplada**: Las sesiones PHP limitan la escalabilidad horizontal.

Este análisis demuestra la importancia de los frameworks modernos, que resuelven estos problemas mediante componentes especializados: routers avanzados, ORMs, sistemas de validación, middlewares, inyección de dependencias, y sistemas de autenticación token-based. Comprender el funcionamiento sin framework es fundamental para utilizar eficazmente estas herramientas y diagnosticar problemas en aplicaciones de producción.

Referencias técnicas

Documentación consultada:

- PHP Manual: PDO, Sessions, password_hash(), filter_var()
- RFC 6749: OAuth 2.0 Authorization Framework (contexto JWT)
- RFC 7519: JSON Web Token (JWT)
- MDN Web Docs: CORS, HTTP Methods, Status Codes
- PostgreSQL Documentation: Prepared Statements