Documento de Arquitectura: API de TimeUp

1. Introducción

Este documento proporciona una guía técnica exhaustiva sobre la arquitectura de la API de TimeUp. Su objetivo es servir como referencia fundamental para los desarrolladores responsables del mantenimiento, la extensión y la comprensión del sistema, detallando los patrones de diseño, los componentes clave y el flujo de datos que rigen su funcionamiento.

La API de TimeUp constituye el núcleo del backend de la aplicación, encargándose de toda la lógica de negocio y la persistencia de datos. Su propósito es gestionar de manera centralizada las entidades principales del sistema, que incluyen usuarios, grupos, calendarios, eventos y tareas, proveyendo una interfaz estructurada para que las aplicaciones cliente puedan interactuar con la información de forma segura y eficiente.

A continuación, se presenta un análisis de la estructura general del sistema, sus componentes y el ciclo de vida de una solicitud.

2. Visión General de la Arquitectura

Comprender la arquitectura de alto nivel es esencial para apreciar cómo las decisiones de diseño impactan directamente en la mantenibilidad, escalabilidad y robustez del proyecto. La elección de un patrón arquitectónico y una clara separación de responsabilidades son la base sobre la cual se construye una aplicación sólida.

2.1. Propósito de la API

El objetivo principal de esta API es actuar como el servicio de backend para la aplicación TimeUp. Se encarga de procesar todas las solicitudes del cliente, ejecutar la lógica de negocio correspondiente, interactuar con la base de datos para persistir y recuperar información, y devolver los datos en un formato estandarizado (JSON) para su consumo.

2.2. Patrón Arquitectónico (MVC Simplificado)

La API sigue una estructura inspirada en el patrón de diseño **Modelo-Vista-Controlador (MVC)**, adaptado para un servicio backend sin una interfaz de usuario tradicional. En este contexto, la "Vista" no es un archivo HTML, sino la respuesta JSON que se envía al cliente.

- **Modelo:** Representa los datos y la lógica de negocio asociada. Cada modelo se corresponde con una tabla de la base de datos (ej. `usuarios`, `eventos`) y encapsula las operaciones de acceso a datos (CRUD: Crear, Leer, Actualizar, Borrar). Su función es interactuar directamente con la base de datos.
- **Controlador:** Actúa como el orquestador. Recibe las peticiones HTTP, interpreta los parámetros, invoca los métodos necesarios en los modelos para manipular los datos y, finalmente, utiliza la "Vista" para formatear la respuesta.
- **Vista (Respuesta JSON):** Es la capa de presentación de los datos. En esta API, su única responsabilidad es formatear los datos obtenidos del controlador en una estructura JSON estándar, lista para ser consumida por un cliente (una aplicación web, móvil, etc.).

2.3. Componentes Principales

El sistema está organizado en varios componentes clave, cada uno con una responsabilidad bien definida:

- **`index.php` (Punto de Entrada / Enrutador):** Es el único punto de acceso público de la API. Todas las solicitudes HTTP son dirigidas a este archivo, que se encarga de interpretar la URL y redirigir la petición al controlador y acción correspondientes.
- **`Controladores`:** Clases que contienen la lógica de la aplicación para cada entidad. Por ejemplo, `UsuarioControlador` maneja el registro y el inicio de sesión de los usuarios. Heredan de `ControladorBase` para reutilizar funcionalidades comunes.
- **`Modelos`:** Clases que representan las tablas de la base de datos y abstraen las operaciones SQL. Por ejemplo, el modelo `Usuario` representa la tabla `usuarios` y hereda de `ModeloBase`, que proporciona métodos CRUD genéricos.
- **`Core`:** Directorio que contiene las clases base fundamentales del framework. Incluye `ModeloBase`, `ControladorBase` y la lógica de conexión a la base de datos (`BaseDeDatos.php`).
- **`config.php` (Configuración):** Archivo centralizado que almacena parámetros del entorno, como las credenciales de conexión a la base de datos. Mantener esta información separada del código mejora la seguridad y facilita la configuración en diferentes entornos (desarrollo, producción).

A continuación, se detallará el flujo exacto que sigue una petición a través de estos componentes para ser procesada.

3. Flujo de una Solicitud HTTP

Entender el ciclo de vida completo de una petición es crucial para depurar errores de manera efectiva y para implementar nuevas funcionalidades de forma coherente con la arquitectura existente. Este conocimiento permite identificar en qué punto de la cadena de ejecución se produce un comportamiento inesperado.

El flujo completo, desde que una solicitud llega al servidor hasta que se devuelve una respuesta, se puede describir en los siguientes pasos:

1. **Recepción en `index.php`:** Una solicitud HTTP, por ejemplo `GET /api/index.php?controlador=usuario&accion=obtener&id=1`, llega al servidor y es manejada por `index.php`. Lo primero que hace el script es configurar las cabeceras HTTP para permitir el acceso desde distintos orígenes (CORS) y definir el tipo de contenido de la respuesta como JSON.
2. **Enrutamiento Básico:** El script `index.php` extrae los parámetros `controlador` y `accion` de la URL utilizando la superglobal `$_GET`. Estos parámetros determinan qué código se debe ejecutar.
3. **Carga del Controlador:** Basándose en el parámetro `controlador`, el script construye dinámicamente el nombre de la clase (ej. `UsuarioControlador`) y la

ruta al archivo correspondiente (ej. `Controllers/UsuarioControlador.php`). A continuación, realiza una serie de validaciones para asegurar que tanto el archivo como la clase existen.

4. **Instanciación y Ejecución:** Si las validaciones son exitosas, se crea una nueva instancia del controlador (ej. `new UsuarioControlador()`). Posteriormente, se invoca el método correspondiente al parámetro `accion` (ej. `$instancia->obtener()`).
5. **Lógica del Controlador:** El método del controlador se ejecuta y procesa la petición. Esto puede implicar leer datos de entrada adicionales (de `$_GET`, `$_POST` o del cuerpo de la petición con `file_get_contents("php://input")`), validar estos datos y coordinar la interacción con el modelo.
6. **Interacción con el Modelo y la BD:** El controlador invoca un método del modelo apropiado para realizar operaciones sobre la base de datos (ej. `Usuario::find(1)`). El modelo, a su vez, utiliza la conexión PDO global establecida en `BaseDeDatos.php` para ejecutar la consulta SQL correspondiente.
7. **Retorno de Datos:** La base de datos devuelve los resultados de la consulta al modelo. El modelo procesa estos resultados y los devuelve al controlador.
8. **Generación de la Respuesta:** El controlador recibe los datos del modelo y utiliza el método `jsonResponse()`, heredado de `ControladorBase`, para formatear la salida. Este método establece el código de estado HTTP (ej. `200 OK`, `404 Not Found`), las cabeceras `Content-Type`, y codifica los datos en formato JSON.
9. **Envío de la Respuesta:** El script finaliza su ejecución con `exit;`, enviando la respuesta JSON completa al cliente que originó la solicitud.

Este flujo estructurado garantiza un procesamiento predecible y coherente de cada solicitud. El siguiente análisis del código de cada componente ilustrará cómo se implementa este proceso en la práctica.

4. Análisis Detallado de Componentes

Para comprender en profundidad cómo funciona la API, es necesario analizar el código fuente de sus componentes principales. Esta sección desglosa la implementación de cada pieza clave para revelar su rol específico y cómo contribuye a la funcionalidad global del sistema.

4.1. El Enrutador ()

El archivo `index.php` es el corazón del sistema de enrutamiento. Su lógica, aunque simple, es fundamental para dirigir el tráfico de la aplicación.

```
// api/index.php
header('Content-Type: application/json; charset=utf-8');
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type, Authorization');
```

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

$controlador = $_GET['controlador'] ?? null;
$accion = $_GET['accion'] ?? null;

// Validar parámetros
if (!$controlador || !$accion) {
    http_response_code(400);
    echo json_encode(['error' => 'Faltan parámetros: controlador o acción']);
    exit;
}

// Construir nombre del controlador
$nombreControlador = ucfirst($controlador) . 'Controlador';
$rutaControlador = __DIR__ . '/Controllers/' . $nombreControlador . '.php';

// Verificar existencia
if (!file_exists($rutaControlador)) {
    http_response_code(404);
    echo json_encode(['error' => 'Controlador no encontrado']);
    exit;
}

// Cargar controlador
require_once $rutaControlador;

// Verificar clase y método
if (!class_exists($nombreControlador)) {
    http_response_code(500);
    echo json_encode(['error' => 'Clase del controlador no encontrada']);
    exit;
}

$instancia = new $nombreControlador();

if (!method_exists($instancia, $accion)) {
    http_response_code(404);
    echo json_encode(['error' => 'Acción no encontrada en el controlador']);
    exit;
}

// Ejecutar acción
try {
    $instancia->$accion();
```

```
} catch (Throwable $e) {
    http_response_code(500);
    echo json_encode(['error' => $e->getMessage()]);
}
```

- **Cabeceras y CORS:** Las primeras líneas establecen las cabeceras HTTP para la respuesta JSON y la política de **Cross-Origin Resource Sharing (CORS)**, permitiendo peticiones desde otros dominios. El bloque `OPTIONS` gestiona correctamente las solicitudes de "pre-vuelo".
- **Extracción de Parámetros:** Se obtienen `controlador` y `accion` de la URL (`$_GET`). El uso del operador `??` es una práctica moderna para manejar la ausencia de parámetros.
- **Validación y Enrutamiento:** El script construye dinámicamente el nombre del controlador y su ruta, seguido de una cadena de validaciones (`file_exists`, `class_exists`, `method_exists`) para asegurar que la solicitud puede ser manejada. Si bien funcional, este mecanismo de enrutamiento acopla la lógica de la aplicación directamente a los parámetros de consulta de la URL, lo cual es frágil y no estándar. Carece de soporte para URLs semánticas y RESTful (ej., `GET /usuarios/42`) y no puede diferenciar entre métodos HTTP para el mismo endpoint, representando una fuente significativa de deuda técnica.
- **Ejecución y Manejo de Errores:** Se instancia el controlador y se invoca la acción. El bloque `try-catch` global es un mecanismo de seguridad crucial que captura excepciones no controladas, previniendo la exposición de errores fatales y asegurando una respuesta JSON formateada.

4.2. El Núcleo ()

El directorio `Core` contiene las clases abstractas y de utilidad que forman el esqueleto de la aplicación.

4.2.1. Conexión a la Base de Datos ()

```
// api/Core/BaseDeDatos.php
// Se encarga únicamente de crear una conexión PDO global
require_once __DIR__ . '/../../config/config.php';

// Cargar configuración
$config = require __DIR__ . '/../../config/config.php';

try {
    // Crear conexión PDO
    $conexion = new PDO(
        "mysql:host={$config['db_host']};dbname={$config['db_name']};charset=utf8mb4",
        $config['db_user'],
        $config['db_pass']
    );
```

```
    // Configurar modo de error para lanzar excepciones
    $conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

} catch (PDOException $e) {
    // Si algo falla, mostrar el error en formato JSON
    http_response_code(500);
    echo json_encode([
        'error' => 'Error de conexión a la base de datos',
        'detalle' => $e->getMessage()
    ]);
    exit;
}
```

Este script tiene una única responsabilidad: establecer la conexión con la base de datos.

- Carga la configuración desde `config.php` y crea una instancia global de `PDO` (`$conexion`) que estará disponible para el resto de la aplicación. Nótese la inclusión redundante de `config.php`; aunque funcionalmente inofensivo debido a `require_once`, esto debe ser limpiado a un único `require` por claridad y buenas prácticas.
- La configuración `PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION` es vital. Obliga a PDO a lanzar excepciones en caso de errores SQL, permitiendo un manejo de errores robusto y centralizado a través de bloques `try-catch`.
- El bloque `try-catch` gestiona fallos de conexión, devolviendo un error 500 en formato JSON, lo cual es una práctica adecuada para fallos críticos de infraestructura.

4.2.2. Controlador Base ()

```
// api/Core/ControladorBase.php
class ControladorBase {
    protected function jsonResponse($data, int $code = 200): void
    {
        http_response_code($code);
        header('Content-Type: application/json; charset=utf-8');
        echo json_encode($data, JSON_UNESCAPED_UNICODE);
        exit;
    }
}
```

Esta clase base proporciona un método de utilidad para estandarizar las respuestas de la API. El método `jsonResponse` centraliza la lógica para establecer el código de estado HTTP, definir la cabecera `Content-Type`, codificar los datos a JSON (`JSON_UNESCAPED_UNICODE` para un manejo correcto de caracteres especiales) y finalizar la ejecución. Todos los controladores deben heredar de esta clase para asegurar la consistencia en las respuestas.

4.2.3. Modelo Base ()

```
// api/Core/ModeloBase.php
// Clase base para todos los modelos del proyecto
require_once __DIR__ . '/BaseDeDatos.php';

abstract class ModeloBase {
    protected static string $tabla = '';
    protected static string $clavePrimaria = '';
    protected array $atributos = [];

    public function __construct(array $data = [])
    {
        $this->atributos = $data;
    }

    public function insert(): int
    {
        global $conexion;
        if (empty($this->atributos)) {
            throw new Exception("No hay datos para insertar.");
        }
        $columnas = array_keys($this->atributos);
        $placeholders = array_map(fn($c) => ':' . $c, $columnas);
        $sql = "INSERT INTO " . static::$tabla .
            " (" . implode(',', $columnas) . ")
            VALUES (" . implode(',', $placeholders) . ")";
        $stmt = $conexion->prepare($sql);
        foreach ($this->atributos as $campo => $valor) {
            $stmt->bindValue(':' . $campo, $valor);
        }
        $stmt->execute();
        return (int) $conexion->lastInsertId();
    }

    public function update(): bool
    {
        global $conexion;
        $idCampo = static::$clavePrimaria;
        if (!isset($this->atributos[$idCampo])) {
            throw new Exception("No se ha definido la clave primaria '$idCampo' en los datos.");
        }
        $id = $this->atributos[$idCampo];
        $columnas = array_filter(array_keys($this->atributos), fn($c) => $c !== $idCampo);
        $setClause = implode(', ', array_map(fn($c) => "$c = :$c", $columnas));
        $sql = "UPDATE " . static::$tabla . " SET $setClause WHERE $idCampo = :$idCampo";
        $stmt = $conexion->prepare($sql);
        foreach ($this->atributos as $campo => $valor) {
```

```
            $stmt->bindValue(':' . $campo, $valor);
        }
        return $stmt->execute();
    }

    public static function deleteById(int $id): bool
    {
        global $conexion;
        $sql = "DELETE FROM " . static::$tabla . " WHERE " . static::$clavePrimaria . " = :id";
        $stmt = $conexion->prepare($sql);
        return $stmt->execute([':id' => $id]);
    }

    public static function all(): array
    {
        global $conexion;
        $sql = "SELECT * FROM " . static::$tabla;
        $stmt = $conexion->query($sql);
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    }

    public static function find(int $id): ?array
    {
        global $conexion;
        $sql = "SELECT * FROM " . static::$tabla . " WHERE " . static::$clavePrimaria . " = :id
LIMIT 1";
        $stmt = $conexion->prepare($sql);
        $stmt->execute([':id' => $id]);
        $res = $stmt->fetch(PDO::FETCH_ASSOC);
        return $res ?: null;
    }
}
```

Esta clase abstracta implementa un patrón **Active Record** simplificado, proporcionando una crucial capa de abstracción que desacopla la lógica de la aplicación en los controladores de la ejecución de SQL puro.

- **Diseño Abstracto:** Obliga a las clases hijas a definir las propiedades estáticas `$tabla` y `$clavePrimaria`, permitiendo que los métodos genéricos funcionen para cualquier entidad.
- **`insert()` y `update()`:** Estos métodos implementan una excelente práctica de seguridad. Construyen consultas SQL dinámicas pero utilizan **sentencias preparadas**. Los métodos recorren en bucle los atributos del modelo, enlazando cada valor individualmente a su marcador de posición correspondiente en la sentencia preparada mediante `bindValue()` antes de la ejecución. Este enfoque previene de manera efectiva los ataques de **inyección SQL**.

- **`all()`, `find()`, `deleteById()`:** Estos métodos estáticos ofrecen una interfaz de alto nivel y muy legible para realizar las operaciones más comunes. Abstraen completamente el SQL subyacente, haciendo que el código en los controladores sea más limpio y declarativo.

4.3. Implementaciones Específicas

Estas clases concretas heredan de las clases base del núcleo y añaden la lógica específica de cada entidad.

4.3.1. Modelo de Usuario ()

```
// api/Models/Usuario.php
require_once __DIR__ . '/../Core/ModeloBase.php';

class Usuario extends ModeloBase {
    protected static string $tabla = 'usuarios';
    protected static string $clavePrimaria = 'id_usuario';

    public static function findByCorreo(string $correo): ?array
    {
        global $conexion;
        $stmt = $conexion->prepare("SELECT * FROM usuarios WHERE correo = :correo
LIMIT 1");
        $stmt->execute([':correo' => $correo]);
        $res = $stmt->fetch(PDO::FETCH_ASSOC);
        return $res ?: null;
    }
}
```

Esta clase es un ejemplo perfecto de cómo extender `ModeloBase`: hereda todos los métodos CRUD, se vincula a la tabla `usuarios` definiendo las propiedades estáticas requeridas y añade un método personalizado, `findByCorreo()`, para implementar una consulta específica de la lógica de negocio de los usuarios.

4.3.2. Controlador de Usuario ()

```
// api/Controllers/UsuarioControlador.php
require_once __DIR__ . '/../Core/ControladorBase.php';
require_once __DIR__ . '/../Core/ModeloBase.php';
require_once __DIR__ . '/../Models/Usuario.php';

class UsuarioControlador extends ControladorBase
{
    // POST /usuario/registrar
    public function registrar()
    {
        $nombre = $_POST['nombre'] ?? null;
```

```
    $correo = $_POST['correo'] ?? null;
    $contrasena = $_POST['contrasena'] ?? null;
    $foto = $_FILES['fotoPerfil'] ?? null;

    if (!$nombre || !$correo || !$contrasena) {
        return $this->jsonResponse(['error' => 'Todos los campos son obligatorios.'], 400);
    }
    if (Usuario::findByCorreo($correo)) {
        return $this->jsonResponse(['error' => 'El correo ya está registrado.'], 409);
    }

    $rutaFoto = null;
    if ($foto && $foto['error'] === UPLOAD_ERR_OK) {
        $nombreArchivo = uniqid('user_') . "_" . basename($foto['name']);
        $rutaDestino = __DIR__ . '/../../public/recursos/perfiles/' . $nombreArchivo;
        if (!file_exists(dirname($rutaDestino))) {
            mkdir(dirname($rutaDestino), 0777, true);
        }
        move_uploaded_file($foto['tmp_name'], $rutaDestino);
        $rutaFoto = 'recursos/perfiles/' . $nombreArchivo;
    }

    $usuario = new Usuario([
        'nombre' => $nombre,
        'correo' => $correo,
        'contrasena' => password_hash($contrasena, PASSWORD_DEFAULT),
        'rol' => 'lector',
        'foto' => $rutaFoto,
        'fecha_creacion' => date('Y-m-d H:i:s')
    ]);

    try {
        $id = $usuario->insert();
        $this->jsonResponse([
            'mensaje' => 'Usuario registrado correctamente',
            'id_usuario' => $id
        ], 201);
    } catch (Throwable $e) {
        $this->jsonResponse(['error' => 'Error al registrar: ' . $e->getMessage()], 500);
    }
}

public function iniciarSesion()
{
    $data = json_decode(file_get_contents("php://input"), true);
    $correo = $data['correo'] ?? '';
    $contrasena = $data['contrasena'] ?? '';
```

```
        if (empty($correo) || empty($contrasena)) {
            echo json_encode(['error' => 'Debe completar todos los campos.']);
            return;
        }

        require_once __DIR__ . '/../Models/Usuario.php';
        $usuario = Usuario::findByCorreo($correo);

        if (!$usuario) {
            echo json_encode(['error' => 'Usuario no encontrado.']);
            return;
        }

        if (!password_verify($contrasena, $usuario['contrasena'])) {
            echo json_encode(['error' => 'Contraseña incorrecta.']);
            return;
        }

        echo json_encode([
            'exito' => true,
            'mensaje' => 'Inicio de sesión correcto',
            'usuario' => [
                'id_usuario' => $usuario['id_usuario'],
                'nombre' => $usuario['nombre'],
                'correo' => $usuario['correo'],
                'rol' => $usuario['rol'],
                'foto' => $usuario['foto'] ?? 'recursos/perfiles/default.png'
            ]
        ]);
    }
}
```

Este controlador orquesta las acciones relacionadas con los usuarios.

- **Análisis de `registrar()`:** Este método sigue correctamente el patrón arquitectónico. Maneja datos `multipart/form-data`, realiza validaciones, ejecuta lógica de negocio (hasheo de contraseñas, subida de archivos), interactúa con el modelo `Usuario` y, crucialmente, utiliza `$this->jsonResponse()` para devolver una respuesta estandarizada.
- **Análisis de `iniciarSesion()`:** Este método, a diferencia del anterior, presenta una **inconsistencia arquitectónica grave**. En lugar de usar el método `jsonResponse()` heredado de `ControladorBase`, recurre a múltiples llamadas directas a `echo json_encode(...)` y `return`. Esto viola el principio de estandarización de respuestas, dificulta el mantenimiento y la refactorización, e introduce un estilo de código divergente. Es un ejemplo claro de por qué la

adherencia a los patrones establecidos en las clases base es fundamental para la salud del proyecto.

Los siguientes ejemplos prácticos ayudarán a consolidar la comprensión de cómo estos componentes colaboran para procesar solicitudes del mundo real.

5. Ejemplos Prácticos de Flujo de Datos

Para materializar la teoría expuesta, esta sección detalla el flujo de datos para dos de los casos de uso más comunes en una API: una operación de lectura (`GET`) para obtener información y una operación de escritura (`POST`) para crear un nuevo recurso.

5.1. Ejemplo de Petición : Obtener un Usuario

Escenario: Un cliente (por ejemplo, una aplicación web) necesita obtener los datos de perfil de un usuario específico para mostrarlos en pantalla.

Petición HTTP: El cliente realiza una petición `GET` a la API, especificando el controlador, la acción y el ID del usuario en la URL.

- `GET /api/index.php?controlador=usuario&accion=obtener&id=42`

Código del Controlador (Hipotético): Basado en las herramientas disponibles en la arquitectura, un método `obtener` en `UsuarioControlador.php` se implementaría de la siguiente manera:

```
// En api/Controllers/UsuarioControlador.php
public function obtener()
{
    $id = $_GET['id'] ?? null;
    if (!$id) {
        return $this->jsonResponse(['error' => 'Falta el ID del usuario.'], 400);
    }

    $usuario = Usuario::find((int)$id);

    if (!$usuario) {
        return $this->jsonResponse(['error' => 'Usuario no encontrado.'], 404);
    }

    // Excluimos la contraseña por seguridad
    unset($usuario['contrasena']);

    $this->jsonResponse($usuario, 200);
}
```

Análisis del Flujo:

1. `index.php` recibe la petición, identifica `controlador=usuario` y `accion=obtener`, y llama al método `UsuarioControlador->obtener()`.
2. El método `obtener()` lee el `id` (42) de la superglobal `$_GET`.
3. Invoca al método estático `Usuario::find(42)`.
4. Como `Usuario` hereda de `ModeloBase`, se ejecuta el método `ModeloBase::find()`, que construye y ejecuta la consulta SQL: `SELECT * FROM usuarios WHERE id_usuario = :id` con `:id` enlazado a `42`.
5. La base de datos devuelve la fila correspondiente al usuario con ID 42.
6. El controlador recibe el array con los datos del usuario, elimina el campo `contrasena` por seguridad y pasa el array resultante al método `jsonResponse()`.

Respuesta JSON: El cliente recibe una respuesta con código `200 OK` y el siguiente cuerpo:


```
{
   "id_usuario": 42,
   "nombre": "Ana López",
   "correo": "ana.lopez@example.com",
   "rol": "lector",
   "foto": "recursos/perfiles/user_...jpg",
   "fecha_creacion": "2023-10-27 10:00:00"
}
```


5.2. Ejemplo de Petición : Registrar un Usuario

Escenario: Un nuevo usuario completa un formulario de registro en la aplicación cliente y envía sus datos para crear una cuenta.

Petición HTTP: El cliente realiza una petición `POST` al endpoint de registro. Los datos se envían típicamente como `form-data` para poder incluir un archivo de imagen.

- `POST /api/index.php?controlador=usuario&accion=registrar`
- **Body (form-data):**
  - `nombre`: "Carlos Ruiz"
  - `correo`: "carlos.ruiz@example.com"
  - `contrasena`: "password123"
  - `fotoPerfil`: (archivo de imagen)

Análisis del Flujo:

1. `index.php` recibe la petición `POST` y la direcciona al método `UsuarioControlador->registrar()`.

2. El controlador extrae los datos de las superglobales `$_POST` (para los campos de texto) y `$_FILES` (para el archivo de imagen).
3. Realiza validaciones: comprueba que los campos no estén vacíos y llama a `Usuario::findByCorreo()` para asegurarse de que el email no esté ya registrado.
4. Si las validaciones son exitosas, hashea la contraseña usando `password_hash()`.
5. Gestiona la subida del archivo, moviendo la imagen desde su ubicación temporal a la carpeta de destino en `public/recursos/perfiles/`.
6. Crea una nueva instancia del modelo: `$usuario = new Usuario([...])`, poblándola con todos los datos procesados (nombre, correo, contraseña hasheada, ruta de la foto, etc.).
7. Invoca al método de instancia `$usuario->insert()`.
8. El método `ModeloBase::insert()` construye dinámicamente la sentencia `INSERT INTO usuarios (nombre, correo, ...) VALUES (:nombre, :correo, ...)` y la ejecuta de forma segura usando sentencias preparadas.
9. La base de datos inserta el nuevo registro y devuelve el ID autoincremental, que es capturado por `$conexion->lastInsertId()`.
10. El controlador recibe este nuevo ID y genera una respuesta de éxito con el código `201 Created`.

Respuesta JSON: El cliente recibe una respuesta que confirma la creación del recurso:

```
{
   "mensaje": "Usuario registrado correctamente",
   "id_usuario": 101
}
```

Estos ejemplos ilustran la robustez y claridad del flujo de trabajo definido por la arquitectura. A continuación, se discutirán áreas donde esta sólida base puede ser mejorada.

6. Buenas Prácticas y Mejoras Sugeridas

La arquitectura actual presenta una base simple y funcional, pero como todo sistema en desarrollo, tiene áreas de mejora significativas. Este análisis crítico y constructivo busca identificar puntos clave para fortalecer la seguridad, el rendimiento y la mantenibilidad de la API a largo plazo.

6.1. Seguridad

6.1.1. Validación de Entradas

La validación actual es mínima. Es imperativo implementar una capa de validación robusta para todos los datos de entrada, verificando tipos, longitudes, formatos (ej. email) y rangos antes de procesarlos.

6.1.2. Prevención de Inyección SQL

El uso sistemático de sentencias preparadas en `ModeloBase` es un punto fuerte de la arquitectura y la medida de seguridad más importante contra inyección SQL. Esta práctica debe mantenerse rigurosamente.

6.1.3. Autenticación y Autorización

La API carece de un mecanismo de autenticación persistente. Se debe implementar un sistema basado en tokens (ej. **JWT**). Tras un login exitoso, se generaría un token que el cliente debe enviar en las cabeceras de las solicitudes a rutas protegidas. Adicionalmente, se debe desarrollar un sistema de autorización basado en el campo `rol` de la tabla `usuarios` para restringir acciones.

6.1.4. Manejo de Errores

Exponer mensajes de error detallados (`$e->getMessage()`) en producción es un riesgo de seguridad. En entornos productivos, el sistema **debe** registrar los errores detallados en un archivo de log inaccesible públicamente y devolver mensajes genéricos al cliente.

6.2. Optimización y Rendimiento

6.2.1. Consultas a la Base de Datos

El método `ModeloBase::all()` es un riesgo de rendimiento, ya que recupera todos los registros de una tabla. Para todos los endpoints que devuelven listas, se debe implementar **paginación** para devolver datos en bloques manejables.

6.2.2. Índices de Base de Datos

El rendimiento de la base de datos debe optimizarse con índices. Por ejemplo, el rendimiento del método `Usuario::findByCorreo()` está garantizado porque la columna `correo` tiene una restricción `UNIQUE`, que crea un índice implícitamente. Sin embargo, si se añade una funcionalidad para buscar todos los eventos de un calendario, la tabla `eventos` **requerirá** un índice explícito en la columna `id_calendario` para evitar escaneos completos de la tabla, que son muy lentos.

6.2.3. Estructura del Proyecto

El uso de `require_once` manuales es propenso a errores. El proyecto **debe** adoptar un **autoloader de Composer (PSR-4)** para eliminar la necesidad de inclusiones manuales, simplificando la gestión de dependencias y la estructura del código.

6.3. Extensibilidad y Mantenibilidad

6.3.1. Enrutamiento Avanzado

El enrutamiento actual basado en parámetros GET es inflexible y representa una deuda técnica significativa. Para garantizar la escalabilidad futura, la aplicación **debe** migrar a un componente de enrutamiento dedicado (como FastRoute). Esto permitiría definir rutas RESTful limpias (ej. `GET /usuarios/{id}`) y reemplazaría toda la lógica de extracción y validación de parámetros en `index.php` con un mapa de rutas declarativo.

6.3.2. Inyección de Dependencias

La dependencia de una conexión global (`global $conexion;`) en los modelos es una mala práctica que genera un acoplamiento fuerte y dificulta las pruebas. La dependencia de una conexión global a la base de datos representa una deuda técnica significativa. Es imperativo refactorizar la capa de acceso a datos para utilizar **inyección de dependencias**, pasando la instancia de PDO a los constructores de los modelos. Esto eliminaría cada instancia de `global $conexion;` en `ModeloBase.php` y `Usuario.php`, desacoplando los componentes y mejorando drásticamente la capacidad de prueba.

6.3.3. Añadir Nuevas Rutas

Actualmente, añadir una nueva funcionalidad requiere crear manualmente un método o un nuevo par de archivos modelo/controlador. Las mejoras sugeridas, como un enrutador avanzado y un autoloader, simplificarían enormemente este proceso, haciéndolo más rápido y menos propenso a errores.

7. Conclusión

La arquitectura de la API de TimeUp se fundamenta en un diseño simple inspirado en el patrón MVC, con una clara separación de responsabilidades y un buen uso de una clase base de modelo para estandarizar el acceso a datos. Su principal fortaleza es la seguridad contra inyecciones SQL gracias al uso correcto de sentencias preparadas.

Sin embargo, su simplicidad introduce debilidades críticas: un enrutamiento frágil, la ausencia total de autenticación y autorización, y una fuerte dependencia de variables globales que constituye una deuda técnica considerable.

Si bien la arquitectura actual es viable para un prototipo, la implementación de las mejoras sugeridas no es opcional, sino **crítica** para el éxito a largo plazo del proyecto. Abordar estas debilidades transformará la API de una base funcional a una plataforma robusta, segura y escalable, preparada para un crecimiento sostenible.