Taller API

Versión 2.0.0

Antonio Martin Sosa

06 de junio de 2025

Contenido:

Bienvenido al sistema de documentación de la API de gestión de vehículos e informes de diagnóstico. Aquí encontrarás toda la información necesaria para entender cómo consumir nuestros endpoints, los modelos de datos y la configuración del proyecto.

Capítulo 1

Introducción

Bienvenido a la documentación técnica de la **API de gestión de vehículos e informes de diagnóstico**. Esta API permite:

- Registrar usuarios y autenticarlos mediante JWT.
- Añadir vehículos a la cuenta de usuario, junto con sus datos OBD-II en tiempo real.
- Almacenar y consultar errores OBD-II (códigos DTC) asociados a cada vehículo.
- Generar informes de diagnóstico y enviarlos por correo a los clientes.
- Compartir informes mediante enlaces públicos protegidos con un token UUID.

1.1. Público Objetivo

- Desarrolladores que estén integrando una solución de telemetría OBD-II.
- Equipos de soporte técnico que necesiten generar y enviar informes de fallos a clientes.
- Administradores de flotas que quieran mantener un registro de diagnósticos remotos.

1.2. Tecnologías Principales

- **FastAPI**: - Framework web para construir APIs RESTful de alto rendimiento. - Documentación automática en Swagger (`/docs`) y ReDoc (`/redoc`).
- **SQLAlchemy**: - ORM para gestionar la base de datos relacional MySQL. - Modelado de entidades como `Usuario`, `Vehiculo`, `ErrorVehiculo` e `InformeCompartido`.
- **Pydantic**: - Validación y serialización de datos en entradas y respuestas.
- **JWT (JSON Web Tokens)**: - Autenticación sin estado (stateless). - Expiración configurable.
- **FastAPI-Mail**: - Envío de correos electrónicos con plantillas HTML para los informes. - Compatible con TLS/SSL y validación de certificados.
- **MySQL**: - Sistema de gestión de base de datos relacional. - Se puede ajustar el motor/host/credenciales a través de `DATABASE_URL`.
- **Sphinx**: - Generación de la documentación estática en formato HTML/Markdown/PDF. - Uso de directivas como `.. automodule::` para extraer docstrings de `main.py`.

1.3. Estructura de la Documentación

Esta documentación se divide en:

1. **Introducción**: visión global, tecnologías y esquema de carpetas.
2. **Endpoints**: descripción detallada de cada ruta, sus parámetros y ejemplos.
3. **Modelos de Datos**: descripción de cada clase ORM y modelo Pydantic.
4. **Configuración**: variables de entorno necesarias y explicación de la configuración del proyecto.

1.4. Requisitos Previos

- Python 3.9+
- MySQL (o servidor compatible)
- Dependencias del proyecto definidas en `requirements.txt` (FastAPI, SQLAlchemy, Pydantic, FastAPI-Mail, python-dotenv, etc.)

Para ejecutar localmente:

1. Clona el repositorio.
2. Crea un entorno virtual:

```
python3 -m venv venv
source venv/bin/activate
```

3. Instala las dependencias del proyecto:

```
pip install -r requirements.txt
```

4. Configura las variables de entorno necesarias creando un archivo `.env` en la raíz del proyecto. Puedes usar como base un archivo `.env.example`.
5. Ejecuta la aplicación usando Uvicorn:

```
uvicorn main:app --reload
```

1.5. Buenas Prácticas

- Mantén las variables de entorno fuera del control de versiones (`.gitignore`).
- Usa HTTPS y certificados válidos en producción.
- Cambia la clave `SECRET_KEY` y el `ALGORITHM` antes de desplegar.
- Implementa control de errores en el frontend para manejar respuestas `401`, `403`, `422`, etc.
- Revisa los logs para rastrear errores en endpoints como `/crear-informe/`.

1.6. Licencia y Contribución

Este proyecto se entrega con fines educativos y está licenciado bajo los términos definidos en el archivo LICENSE.

Si deseas contribuir:

- Realiza un fork del repositorio.
- Crea una rama para tu mejora o corrección:

```
git checkout -b feature/nombre
```

- Haz tus commits siguiendo buenas prácticas de formato y mensajes.
- Envía un Pull Request para revisión.

1.7. Contacto

Para dudas, errores o sugerencias, puedes abrir un issue en GitHub o contactar directamente con el equipo de desarrollo a traves del repositorio de GitHub

Capítulo 2

Endpoints de la API

En este documento se describen todos los endpoints disponibles en la API, organizados por funcionalidad. Para cada ruta se indica método HTTP, URL, esquema de datos de entrada/salida, validaciones y ejemplos de uso.

2.1. Autenticación y Usuarios

2.1.1. Registro de Usuario

`POST /register`

Registra un nuevo usuario en la base de datos.

Request Body (`UsuarioRegistro`):

- `username` (string, obligatorio, mínimo 3 caracteres)
- `password` (string, obligatorio, mínimo 6 caracteres)

Ejemplo:

```
POST /register HTTP/1.1
Host: api.ejemplo.com
Content-Type: application/json

{
  "username": "juanperez",
  "password": "secreto123"
}
```

Responses:

- **200 OK**

```
{
  "mensaje": "Usuario registrado correctamente"
}
```

- **400 Bad Request**
  - Cuando `username` o `password` no cumplen longitud mínima.

- Si el `username` ya existe.

```
{
  "detail": "El usuario ya existe"
}
```

- **500 Internal Server Error**
  - Error inesperado al guardar en la base de datos.

2.1.2. Login / Obtención de Token

`POST /login`

Autentica al usuario y devuelve un token JWT.

Request Body (`UsuarioLogin`):

- `username` (string, obligatorio)
- `password` (string, obligatorio)

Ejemplo:

```
POST /login HTTP/1.1
Host: api.ejemplo.com
Content-Type: application/json

{
  "username": "juanperez",
  "password": "secreto123"
}
```

Responses:

- **200 OK**

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer"
}
```

- **400 Bad Request**
  - Si `username` o `password` no cumplen requisitos (longitud).

```
{
  "detail": "Las credenciales no cumplen los requisitos mínimos"
}
```

- **401 Unauthorized**
  - Usuario no registrado o contraseña incorrecta.

```
{
  "detail": "Credenciales inválidas"
}
```

- **500 Internal Server Error**
  - Error al generar el token.

2.2. Gestión de Vehículos

> **Nota**
>
> Todos los endpoints de esta sección requieren que el header `Authorization: Bearer <token>` contenga un JWT válido.

2.2.1. Guardar Vehículo

`POST /guardar-vehiculo/`

Crea un nuevo vehículo y lo asocia al usuario autenticado.

Request Body (`VehiculoRegistro`):

- `marca` (string, obligatorio)
- `modelo` (string, obligatorio)
- `year` (integer, obligatorio, p. ej., 2023)
- `rpm` (integer, obligatorio)
- `velocidad` (integer, obligatorio)
- `vin` (string, obligatorio, longitud = 17)
- `revision` (dict JSON, obligatorio) - `tipo` (string, obligatorio) - `fecha` (string, formato ISO YYYY-MM-DD, obligatorio) - `observaciones` (string, opcional)

Ejemplo:

```
POST /guardar-vehiculo/ HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
Content-Type: application/json

{
  "marca": "Ford",
  "modelo": "Focus",
  "year": 2020,
  "rpm": 1200,
  "velocidad": 60,
  "vin": "1HGCM82633A004352",
  "revision": {
    "tipo": "General",
    "fecha": "2025-06-05",
    "observaciones": "Cambio aceite"
  }
}
```

Responses:

- **200 OK**

```
{
  "mensaje": "Vehículo guardado correctamente",
  "id": 10
}
```

- **400 Bad Request**
  - VIN inválido (no 17 caracteres) o campos faltantes.
  - VIN duplicado.

```
{
  "detail": "VIN inválido o ya registrado"
}
```

- **401 Unauthorized**
  - Token ausente, inválido o expirado.

```
{
  "detail": "No autenticado"
}
```

- **500 Internal Server Error**
  - Error interno al persistir datos.

2.2.2. Listar Vehículos del Usuario

`GET /mis-vehiculos/`

Recupera todos los vehículos asociados al usuario autenticado.

Headers:

- `Authorization: Bearer <token>` (string, obligatorio)

Ejemplo:

```
GET /mis-vehiculos/ HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
```

Responses:

- **200 OK**
  - Si existen vehículos:

```
{
  "vehiculos": [
    {
      "id": 1,
      "marca": "Toyota",
      "modelo": "Corolla",
```

(continúe en la próxima página)

(proviene de la página anterior)


```
      "year": 2018,
      "rpm": 1500,
      "velocidad": 80,
      "vin": "JTDBL40E799017833",
      "revision": {"tipo": "Anual", "fecha": "2025-01-10"},
      "usuario_id": 5
    },
    {
      "id": 2,
      "marca": "Honda",
      "modelo": "Civic",
      "year": 2021,
      "rpm": 1300,
      "velocidad": 70,
      "vin": "2HGFC2F59MH123456",
      "revision": {"tipo": "Cambio llantas", "fecha": "2025-03-20"},
      "usuario_id": 5
    }
  ]
}
```


- Si no hay vehículos:


```
{
  "mensaje": "No hay vehículos registrados para este usuario.",
  "vehiculos": []
}
```


- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error al recuperar datos.

2.2.3. Obtener Vehículo Específico

`GET /mis-vehiculos/{vehiculo_id}`

Recupera la información de un único vehículo (`vehiculo_id`) si pertenece al usuario.

URL Parameters:

- `vehiculo_id` (integer, obligatorio)

Ejemplo:

```
GET /mis-vehiculos/2 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
```

Responses:

- **200 OK**

```
{
  "id": 2,
  "marca": "Honda",
  "modelo": "Civic",
  "year": 2021,
  "rpm": 1300,
  "velocidad": 70,
  "vin": "2HGFC2F59MH123456",
  "revision": {"tipo": "Cambio llantas", "fecha": "2025-03-20"},
  "usuario_id": 5
}
```


- **404 Not Found**
  - El vehículo no existe o no pertenece al usuario.


```
{
  "detail": "Vehículo no encontrado"
}
```


- **401 Unauthorized**
  - Token inválido o expirado.

2.2.4. Editar Vehículo

`PUT /editar-vehiculo/{vehiculo_id}`

Actualiza los datos de un vehículo existente.

URL Parameters:

- `vehiculo_id` (integer, obligatorio)

Request Body (`VehiculoEdicion`):

- `marca` (string, obligatorio)
- `modelo` (string, obligatorio)
- `year` (integer, obligatorio)
- `rpm` (integer, obligatorio)
- `velocidad` (integer, obligatorio)
- `vin` (string, obligatorio, longitud = 17)

Ejemplo:

```
PUT /editar-vehiculo/2 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
Content-Type: application/json

{
  "marca": "Honda",
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  "modelo": "Civic LX",
  "year": 2022,
  "rpm": 1400,
  "velocidad": 75,
  "vin": "2HGFC2F59MH123456"
}
```

Responses:

- **200 OK**

```
{
  "mensaje": "Vehículo actualizado correctamente"
}
```

- **400 Bad Request**
  - VIN inválido o ya registrado en otro vehículo.

```
{
  "detail": "VIN duplicado"
}
```

- **404 Not Found**
  - Vehículo no existe o no pertenece al usuario.

```
{
  "detail": "Vehículo no encontrado"
}
```

- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error interno al actualizar.

2.2.5. Eliminar Vehículo

`DELETE /eliminar-vehiculo/{vehiculo_id}`

Elimina un vehículo y todos sus errores OBD-II asociados (cascade).

URL Parameters:

- `vehiculo_id` (integer, obligatorio)

Ejemplo:

```
DELETE /eliminar-vehiculo/2 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
```

Responses:

- **200 OK**

```
{
  "mensaje": "Vehículo eliminado correctamente",
  "errores_eliminados": 4
}
```

  - `errores_eliminados`: cantidad de registros de errores borrados.

- **404 Not Found**
  - Vehículo no existe o no pertenece al usuario.

```
{
  "detail": "Vehículo no encontrado"
}
```

- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error interno al eliminar.

2.3. Gestión de Errores OBD-II

> **Nota**
>
> Estos endpoints también requieren token válido en `Authorization`.

2.3.1. Guardar Errores de Vehículo

`POST /guardar-errores/`

Registra múltiples códigos DTC para un vehículo del usuario.

Request Body (`ErrorVehiculoRegistro`):

- `vehiculo_id` (integer, obligatorio)
- `codigo_dtc` (array[string], obligatorio) - Lista de códigos OBD-II (p. ej.: `["P0301", "P0420", "P0133"]`)

Validaciones:

- `vehiculo_id` debe ser entero positivo y corresponder a un vehículo del usuario.
- `codigo_dtc` no puede estar vacío ni contener duplicados o valores en blanco.

Ejemplo:

```
POST /guardar-errores/ HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
```

(continúe en la próxima página)

(proviene de la página anterior)

```
Content-Type: application/json

{
  "vehiculo_id": 1,
  "codigo_dtc": ["P0301", "P0420", "P0171"]
}
```

Responses:

- **200 OK**

```
{
  "mensaje": "Errores del vehículo guardados correctamente"
}
```

- **400 Bad Request**
  - Formato inválido, lista vacía, duplicados, vehiculo_id negativo.

```
{
  "detail": "Lista de códigos vacía o contiene duplicados"
}
```

- **404 Not Found**
  - Vehículo no encontrado o no pertenece al usuario.

```
{
  "detail": "Vehículo no encontrado"
}
```

- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error interno al guardar.

2.3.2. Obtener Errores de un Vehículo

`GET /mis-errores/{vehiculo_id}`

Obtiene todos los códigos DTC registrados para un vehículo.

URL Parameters:

- `vehiculo_id` (integer, obligatorio)

Ejemplo:

```
GET /mis-errores/1 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
```

Responses:

- **200 OK**

```
["P0301", "P0420", "P0171"]
```

- **404 Not Found**
  - Vehículo no existente o no tiene errores registrados.

```
{
  "detail": "No se encontraron errores para este vehículo"
}
```

- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error interno al consultar la base de datos.

2.4. Generación y Consulta de Informes

Nota

La creación del informe envía un correo con un enlace público (token UUID). Ver también en "Modelos" la tabla `InformeCompartido`.

2.4.1. Crear Informe para un Vehículo

`POST /crear-informe/{vehiculo_id}`

Genera un informe HTML con todos los errores de un vehículo y lo envía por correo al cliente. Se crea un token único para acceso público.

URL Parameters:

- `vehiculo_id` (integer, obligatorio)

Request Body (`InformeRequest`):

- `email` (string, obligatorio, debe contener «@»).

Flujo Interno:

1. Verificar que `vehiculo_id` existe y pertenece al usuario.
2. Recuperar lista de códigos DTC asociados.
3. Generar un token UUID único (`uuid4()`) y guardar en tabla `informes_compartidos`.
4. Crear plantilla HTML (botón con enlace a `/informe/{token}`).
5. Enviar correo con FastAPI-Mail.

Ejemplo:

```
POST /crear-informe/1 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer eyJhbGciOi...
Content-Type: application/json

{
  "email": "cliente@dominio.com"
}
```

Responses:

- **200 OK**

```
{
  "mensaje": "Informe creado y enviado al email",
  "token": "550e8400-e29b-41d4-a716-446655440000",
  "enlace": "https://tudominio.com/taller-front/informe/550e8400-e29b-
↪41d4-a716-446655440000"
}
```

- **400 Bad Request**
  - Email inválido (no contiene «@») o faltan datos.

```
{
  "detail": "Formato de email inválido"
}
```

- **404 Not Found**
  - Vehículo no existe o no pertenece al usuario.

```
{
  "detail": "Vehículo no encontrado"
}
```

- **401 Unauthorized**
  - Token inválido o expirado.
- **500 Internal Server Error**
  - Error al guardar token o enviar correo.

2.5. Ver Informe Público

`GET /informe/{token}`

Permite a cualquier usuario (sin autenticación) ver el informe de diagnóstico de un vehículo, siempre que posea el token correcto.

URL Parameters:

- `token` (string, obligatorio, longitud mínima 10)

Flujo Interno:

1. Validar longitud mínima de `token` (10 caracteres).
2. Buscar registro en tabla `informes_compartidos`.
3. Recuperar datos de vehículo e historial de errores.
4. Devolver JSON con datos del vehículo y lista de errores.

Ejemplo:

```
GET /informe/550e8400-e29b-41d4-a716-446655440000 HTTP/1.1
Host: api.ejemplo.com
```

Responses:

- **200 OK**

```
{
  "vehiculo": {
    "marca": "Toyota",
    "modelo": "Corolla",
    "year": 2020,
    "vin": "JTDBL40E799017833",
    "rpm": 1500,
    "velocidad": 80,
    "revision": {"tipo": "Anual", "fecha": "2025-01-10"}
  },
  "errores": ["P0301", "P0420"]
}
```

- **400 Bad Request**
  - Token inválido (longitud < 10).

```
{
  "detail": "Token inválido"
}
```

- **404 Not Found**
  - Token no encontrado o informe expirado.

```
{
  "detail": "Informe no encontrado"
}
```

- **500 Internal Server Error**
  - Error interno al procesar la solicitud.

2.6. Servicio de Imágenes de Vehículo

2.6.1. Obtener URL de Imagen de Vehículo

`GET /car-imagery/`

Recupera una URL de imagen de vehículo usando la API externa de **carimagery.com**.

Query Parameters:

- `searchTerm` (string, obligatorio) Ejemplo: `searchTerm=Toyota%20Corolla%202020`

Ejemplo:

```
GET /car-imagery/?searchTerm=Toyota%20Corolla%202020 HTTP/1.1
Host: api.ejemplo.com
```

Responses:

- **200 OK**

  Devuelve texto o XML con la URL de la imagen, por ejemplo:

  ```
  <CarImagery>
    <ImageUrl>https://imagenes.com/ford_focus_2020.jpg</ImageUrl>
  </CarImagery>
  ```

- **400 Bad Request**

  - Si falta `searchTerm` o está vacío.

  ```
  {
    "detail": "searchTerm es obligatorio"
  }
  ```

- **500 Internal Server Error**

  - Error al conectar con la API externa.

2.7. Endpoint de Prueba / Salud

2.7.1. Salud / Ping

`GET /saludo`

Endpoint simple para verificar que la API está en funcionamiento.

Ejemplo:

```
GET /saludo HTTP/1.1
Host: api.ejemplo.com
```

Responses:

- **200 OK**

  ```
  {
    "mensaje": "¡La API está funcionando correctamente!"
  }
  ```

Capítulo 3

Modelos de Datos

En esta sección se describen en detalle los **modelos ORM** (SQLAlchemy) y los **modelos de validación** (Pydantic) definidos en *main.py*. Se incluyen tablas con atributos, tipos, relaciones y ejemplos de uso.

3.1. Modelos ORM (SQLAlchemy)

Los modelos ORM representan las tablas de la base de datos, heredan de *Base* y están definidos en *main.py*. A continuación se listan sus atributos principales, tipos y descripciones.

Clase	Atributo	Tipo	Descripción
Usuario	id	Integer (PK)	Identificador único autoincremental.
Usuario	username	String(255), único	Nombre de usuario (único).
Usuario	password_hash	String(255)	Contraseña almacenada (hash bcrypt).
Usuario	vehiculos	Relationship	Relación uno-a-muchos con Vehiculo.
Vehiculo	id	Integer (PK)	Identificador del vehículo.
Vehiculo	marca	String(255)	Marca del vehículo.
Vehiculo	modelo	String(255)	Modelo del vehículo.
Vehiculo	year	Integer	Año de fabricación (YYYY).
Vehiculo	rpm	Integer	Revoluciones por minuto.
Vehiculo	velocidad	Integer	Velocidad actual en km/h.
Vehiculo	vin	String(17), único	Número VIN (único, 17 caracteres).
Vehiculo	revision	String(255)	JSON serializado con detalles de la revisión.
Vehiculo	usuario_id	Integer (FK)	Clave foránea a Usuario.id.
Vehiculo	errores	Relationship	Relación uno-a-muchos con ErrorVehiculo.
Vehiculo	informes_compartidos	Relationship	Relación uno-a-muchos con InformeCompartido.
ErrorVehiculo	id	Integer (PK)	Identificador del error DTC.
ErrorVehiculo	vehiculo_id	Integer (FK)	Clave foránea a Vehiculo.id.
ErrorVehiculo	codigo_dtc	String(255)	Código OBD-II (p. ej., P0301).
ErrorVehiculo	vehiculo	Relationship	Vínculo inverso a Vehiculo.
InformeCompart	id	Integer (PK)	Identificador del informe.
InformeCompart	token	String(100), único	Token UUID público para compartir.
InformeCompart	vehiculo_id	Integer (FK)	Clave foránea a Vehiculo.id.
InformeCompart	email_cliente	String(255)	Email del destinatario.
InformeCompart	creado_en	DateTime	Fecha y hora de creación (UTC).

Relaciones entre Tablas:

- Usuario ↔ Vehiculo
  - Uno a muchos:
    - En Usuario:

```
vehiculos = relationship("Vehiculo", back_populates="usuario",␣
↪cascade="all, delete-orphan")
```

    - En Vehiculo:

```
usuario = relationship("Usuario", back_populates="vehiculos")
```

- Vehiculo ↔ ErrorVehiculo
  - Uno a muchos:

- En **Vehiculo**:

```
errores = relationship("ErrorVehiculo", back_populates="vehiculo",␣
↪cascade="all, delete-orphan")
```

- En **ErrorVehiculo**:

```
vehiculo = relationship("Vehiculo", back_populates="errores")
```

- **Vehiculo** ↔ **InformeCompartido**
  - Uno a muchos:
    - En **Vehiculo**:

```
informes_compartidos = relationship("InformeCompartido", back_
↪populates="vehiculo", cascade="all, delete-orphan")
```

    - En **InformeCompartido**:

```
vehiculo = relationship("Vehiculo", back_populates="informes_
↪compartidos")
```

3.1.1. Ejemplo de Esquema en SQL

A modo de referencia, a continuación se muestra un esquema simplificado en SQL que refleja la estructura anterior:

```
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL
);

CREATE TABLE vehiculos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    marca VARCHAR(255) NOT NULL,
    modelo VARCHAR(255) NOT NULL,
    year INT NOT NULL,
    rpm INT NOT NULL,
    velocidad INT NOT NULL,
    vin VARCHAR(17) UNIQUE NOT NULL,
    revision TEXT NOT NULL,
    usuario_id INT NOT NULL,
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id) ON DELETE CASCADE
);

CREATE TABLE error_vehiculo (
    id INT AUTO_INCREMENT PRIMARY KEY,
    vehiculo_id INT NOT NULL,
    codigo_dtc VARCHAR(255) NOT NULL,
    FOREIGN KEY (vehiculo_id) REFERENCES vehiculos(id) ON DELETE CASCADE
```

(continúe en la próxima página)

(proviene de la página anterior)

```
);

CREATE TABLE informes_compartidos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    token VARCHAR(100) UNIQUE NOT NULL,
    vehiculo_id INT NOT NULL,
    email_cliente VARCHAR(255) NOT NULL,
    creado_en DATETIME NOT NULL,
    FOREIGN KEY (vehiculo_id) REFERENCES vehiculos(id) ON DELETE CASCADE
);
```

3.2. Modelos de Validación (Pydantic)

Los modelos Pydantic garantizan que las peticiones entrantes y salidas cumplan con un esquema específico. Se utilizan en los endpoints para validar datos de usuario, vehículo, errores e informes.

Clase	Campos	Descripción
`UsuarioRegist`	`username` (string, obligatorio, mínimo 3 caracteres)	Nombre de usuario.
`UsuarioRegist`	`password` (string, obligatorio, mínimo 6 caracteres)	Contraseña.
`UsuarioLogin`	`username` (string, obligatorio)	Nombre de usuario.
`UsuarioLogin`	`password` (string, obligatorio)	Contraseña.
`VehiculoRegis`	`marca` (string, obligatorio)	Marca del vehículo.
`VehiculoRegis`	`modelo` (string, obligatorio)	Modelo del vehículo.
`VehiculoRegis`	`year` (integer, obligatorio, YYYY)	Año de fabricación.
`VehiculoRegis`	`rpm` (integer, obligatorio)	Revoluciones por minuto.
`VehiculoRegis`	`velocidad` (integer, obligatorio)	Velocidad actual en km/h.
`VehiculoRegis`	`vin` (string de longitud 17, obligatorio)	Número VIN único.
`VehiculoRegis`	`revision` (dict con claves obligatorias y opcionales)	Detalles de revisión: `tipo`, `fecha`, `observaciones`.
`VehiculoEdici`	(mismos campos que `VehiculoRegistro` excepto `revision`)	Usado para actualizar vehículos existentes.
`ErrorVehiculc`	`vehiculo_id` (integer, obligatorio)	ID del vehículo.
`ErrorVehiculc`	`codigo_dtc` (list[string], obligatorio)	Lista de códigos DTC.
`InformeReques`	`email` (string, obligatorio, debe contener «@»)	Email del cliente.

3.2.1. Validaciones Clave

- En `UsuarioRegistro` y `UsuarioLogin` se verifica longitud mínima de *username* y *password*.
- En `VehiculoRegistro` y `VehiculoEdicion` se exige que *vin* sea exactamente 17 caracteres y único en la base de datos.

- En `ErrorVehiculoRegistro` se valida que *vehiculo_id* exista y que la lista *codigo_dtc* no esté vacía ni tenga duplicados.
- En `InformeRequest` se comprueba que *email* contenga el carácter "@".

3.2.2. Ejemplo de Uso en un Endpoint

Por ejemplo, en el endpoint */guardar-vehiculo/*:

```
@app.post("/guardar-vehiculo/")
def crear_vehiculo(
    vehiculo: VehiculoRegistro,
    db: Session = Depends(get_db),
    usuario: Usuario = Depends(obtener_usuario_desde_token)
):
    # Aquí 'vehiculo' ya está validado por Pydantic:
    #   vehiculo.marca (str), vehiculo.modelo (str), vehiculo.vin (str de 17␣
↪chars), etc.
    nuevo = models.Vehiculo(
        marca=vehiculo.marca,
        modelo=vehiculo.modelo,
        year=vehiculo.year,
        rpm=vehiculo.rpm,
        velocidad=vehiculo.velocidad,
        vin=vehiculo.vin,
        revision=json.dumps(vehiculo.revision),
        usuario_id=usuario.id
    )
    db.add(nuevo)
    db.commit()
    db.refresh(nuevo)
    return {"mensaje": "Vehículo guardado correctamente", "id": nuevo.id}
```

3.2.3. Referencia Automática

Para revisar el código completo de cada clase (atributos adicionales, métodos, relaciones, validaciones), se ha añadido la directiva *.. automodule:: main* más abajo:

`class main.Base(`***kwargs: Any*`)`

Bases: `object`

Configuración del sistema de envío de correos (FastAPI Mail):

- Las credenciales y parámetros se cargan desde variables de entorno.
- *FastMail* se instancia con esta configuración para ser usado en envíos.

`metadata: MetaData = MetaData()`

`registry: registry = <sqlalchemy.orm.decl_api.registry object>`

`class main.ErrorVehiculo(`***kwargs*`)`

Bases: `Base`

Modelo ORM que almacena los errores OBD-II (códigos DTC) de un vehículo.

Atributos:
id (int): ID del error. vehiculo_id (int): ID del vehículo asociado. codigo_dtc (str): Código de diagnóstico (ej. P0301).

Relaciones:
vehiculo (Vehiculo): Vehículo asociado.

`codigo_dtc`

`id`

`vehiculo`

`vehiculo_id`

`class main.ErrorVehiculoRegistro(`** (Keyword-only parameters separator (PEP 3102))*, *codigo_dtc: list[str]*, *vehiculo_id: int*`)`

Bases: `BaseModel`

Modelo de solicitud para registrar errores OBD-II de un vehículo.

Atributos:
codigo_dtc (list[str]): Lista de códigos DTC (códigos de diagnóstico). vehiculo_id (int): ID del vehículo al que se le asocian los errores.

`codigo_dtc: list[str]`

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`vehiculo_id: int`

`class main.InformeCompartido(`***kwargs*`)`

Bases: `Base`

Modelo ORM que representa un informe compartido con un cliente por email.

Atributos:
id (int): ID del informe. token (str): Token único para acceder al informe. vehiculo_id (int): ID del vehículo relacionado. email_cliente (str): Email al que se envía el informe. creado_en (str): Fecha y hora de creación del informe (ISO format).

Relaciones:
vehiculo (Vehiculo): Vehículo asociado.

`creado_en`

`email_cliente`

`id`

`token`

`vehiculo`

`vehiculo_id`

class main.InformeRequest(*, *email: str*)

Bases: BaseModel

Modelo de solicitud para generar y enviar un informe por correo.

Atributos:

email (str): Dirección de email del cliente destinatario.

email: str

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class main.Usuario(**kwargs)

Bases: Base

Modelo ORM que representa a los usuarios del sistema.

Atributos:

id (int): ID autoincremental (clave primaria). username (str): Nombre de usuario, único. password_hash (str): Contraseña hasheada con bcrypt.

Relaciones:

vehiculos (List[Vehiculo]): Lista de vehículos registrados por el usuario.

id

password_hash

username

vehiculos

class main.UsuarioLogin(*, *username: str*, *password: str*)

Bases: BaseModel

Modelo de solicitud para iniciar sesión de usuario.

Atributos:

username (str): Nombre de usuario. password (str): Contraseña en texto plano.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

password: str

username: str

class main.UsuarioRegistro(*, *username: str*, *password: str*)

Bases: BaseModel

Modelo de solicitud para registrar un nuevo usuario.

Atributos:

username (str): Nombre de usuario. password (str): Contraseña en texto plano.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

password: str

username: str

class main.Vehiculo(**kwargs)

Bases: Base

Modelo ORM que representa un vehículo registrado.

Atributos:

id (int): ID del vehículo. marca (str): Marca del vehículo. modelo (str): Modelo del vehículo. year (int): Año de fabricación. rpm (int): Revoluciones por minuto. velocidad (int): Velocidad actual. vin (str): Número VIN único del vehículo. revision (str): Información de revisión técnica. usuario_id (int): ID del usuario al que pertenece el vehículo.

Relaciones:

usuario (Usuario): Usuario propietario. errores (List[ErrorVehiculo]): Lista de errores asociados. informes_compartidos (List[InformeCompartido]): Informes generados con token público.

errores

id

informes_compartidos

marca

modelo

revision

rpm

usuario

usuario_id

velocidad

vin

year

class main.VehiculoEdicion(*, *marca: str*, *modelo: str*, *year: int*, *rpm: int*, *velocidad: int*, *vin: str*)

Bases: BaseModel

Modelo de solicitud para editar un vehículo existente.

Atributos:

marca (str): Marca del vehículo. modelo (str): Modelo del vehículo. year (int): Año de fabricación. rpm (int): Revoluciones por minuto. velocidad (int): Velocidad actual. vin (str): Número VIN del vehículo.

marca: str

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

modelo: str

rpm: int

velocidad: int

vin: str

year: int

class main.VehiculoRegistro(*, *marca: str*, *modelo: str*, *year: int*, *rpm: int*, *velocidad: int*, *vin: str*, *revision: dict*)

Bases: BaseModel

Modelo de solicitud para registrar un nuevo vehículo.

Atributos:

marca (str): Marca del vehículo. modelo (str): Modelo del vehículo. year (int): Año del vehículo. rpm (int): RPM del motor. velocidad (int): Velocidad del vehículo. vin (str): Número VIN único del vehículo. revision (dict): Detalles de la revisión técnica (estructura flexible).

marca: str

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

modelo: str

revision: dict

rpm: int

velocidad: int

vin: str

year: int

async main.crear_informe(*vehiculo_id: int*, *request:* InformeRequest, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*)

Crea un informe de errores del vehículo y lo envía al email del cliente.

Este endpoint genera un enlace único que da acceso a una vista del informe de diagnóstico del vehículo. Se envía un correo al cliente con dicho enlace.

Parámetros

- vehiculo_id (int) – ID del vehículo del que se desea generar el informe.

- `request` (`InformeRequest`) – Objeto que contiene el email del cliente.
- `usuario` (`Usuario`) – Usuario autenticado mediante JWT.
- `db` (`Session`) – Sesión activa de la base de datos.

Devuelve
Mensaje de éxito, token generado y enlace de acceso.

Tipo del valor devuelto
dict

Muestra

- `HTTPException 400` – Si el email no es válido.
- `HTTPException 404` – Si el vehículo no pertenece al usuario.
- `HTTPException 500` – Si ocurre un error al guardar el informe o enviar el correo.

`main.crear_token(`*data: dict*, *expira_en: int = 300*`)`

Genera un token JWT con los datos proporcionados y un tiempo de expiración opcional.

Parámetros

- `datos` (`dict`) – Datos a incluir en el payload del token.
- `tiempo_expiracion` (`Optional[timedelta]`) – Tiempo personalizado de expiración. Si no se especifica, se usarán 30 minutos por defecto.

Devuelve
Token JWT firmado.

Tipo del valor devuelto
str

Muestra
`Exception` – Si hay un error al codificar el token.

`main.editar_vehiculo(`*vehiculo_id: int*, *datos:* VehiculoEdicion, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*`)`

Actualiza los datos de un vehículo existente del usuario autenticado.

Parámetros

- `vehiculo_id` (`int`) – ID del vehículo a modificar.
- `datos_actualizados` (`VehiculoBase`) – Nuevos datos del vehículo.
- `db` (`Session`) – Sesión de base de datos.
- `usuario` (`Usuario`) – Usuario autenticado.

Devuelve
Mensaje de éxito.

Tipo del valor devuelto
dict

Muestra
`HTTPException 404` – Si el vehículo no existe o no pertenece al usuario.

`main.eliminar_vehiculo(`*vehiculo_id: int*, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*`)`

Elimina un vehículo registrado por el usuario autenticado.

Parámetros

- `vehiculo_id` (`int`) – ID del vehículo a eliminar.
- `db` (`Session`) – Sesión de base de datos.
- `usuario` (`Usuario`) – Usuario autenticado mediante JWT.

Devuelve
Mensaje de éxito.

Tipo del valor devuelto
dict

Muestra
`HTTPException 404` – Si el vehículo no existe o no pertenece al usuario.

`main.fm = <fastapi_mail.fastmail.FastMail object>`

Configuración de seguridad:

- *SECRET_KEY*, *ALGORITHM* y tiempo de expiración definen la seguridad del JWT.
- *pwd_context* se usa para hashear contraseñas con bcrypt.
- *oauth2_scheme* se usa como dependencia para extraer el token del header Authorization.

`main.get_car_image(`*searchTerm: str*`)`

Obtiene una URL de imagen representativa de un vehículo usando el término de búsqueda proporcionado.

Este endpoint consulta la API externa de carimagery.com para devolver la URL de una imagen que coincida con el término (por ejemplo, «Toyota Corolla 2020»).

Parámetros
`searchTerm` (`str`) – Término de búsqueda del vehículo (marca, modelo, año, etc.).

Devuelve
URL de la imagen del vehículo.

Tipo del valor devuelto
str

Muestra
`HTTPException 500` – Si hay un error al consultar la API externa.

`main.guardar_errores(`*datos:* ErrorVehiculoRegistro, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*`)`

Guarda una lista de códigos de error OBD-II (DTC) asociados a un vehículo del usuario autenticado.

Este endpoint es utilizado por el cliente Python que recibe errores del escáner OBD-II y los envía al backend para su almacenamiento.

Parámetros

- **datos** (**ErrorVehiculoRegistro**) – Objeto que contiene el ID del vehículo y una lista de códigos DTC.
- **usuario** (**Usuario**) – Usuario autenticado, obtenido desde el token JWT.
- **db** (**Session**) – Sesión activa de la base de datos.

Devuelve

Mensaje de confirmación si los errores fueron guardados correctamente.

Tipo del valor devuelto

dict

Muestra

- **HTTPException 400** –
  - Si el ID del vehículo no es válido (no entero o negativo). - Si la lista de códigos está vacía o contiene valores vacíos. - Si hay códigos DTC duplicados.
- **HTTPException 404** – Si el vehículo no pertenece al usuario autenticado.
- **HTTPException 500** – Si ocurre un error inesperado al guardar en la base de datos.

main.guardar_vehiculo(*datos:* VehiculoRegistro, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*)

Guarda un nuevo vehículo en la base de datos asociado al usuario autenticado.

Parámetros

- **vehiculo** (**VehiculoBase**) – Datos del vehículo (marca, modelo, año, color, etc.).
- **db** (**Session**) – Sesión de base de datos.
- **usuario** (**Usuario**) – Usuario autenticado, extraído desde el token JWT.

Devuelve

Mensaje de confirmación.

Tipo del valor devuelto

dict

Muestra

HTTPException 401 – Si no se proporciona un token válido.

main.login(*datos:* UsuarioLogin, *db: Session = Depends(get_db)*)

Autentica al usuario y devuelve un token JWT válido.

Parámetros

- **datos** (**UsuarioLogin**) – Credenciales de usuario.

- **db** (**Session**) – Sesión activa de la base de datos.

Devuelve
Token JWT si la autenticación fue exitosa.

Tipo del valor devuelto
dict

Muestra

- **HTTPException 400** – Datos inválidos.
- **HTTPException 401** – Usuario no encontrado o contraseña incorrecta.
- **HTTPException 500** – Error al generar el token.

main.obtener_errores(*vehiculo_id: int*, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*)

Devuelve todos los errores DTC (códigos OBD-II) asociados a un vehículo del usuario autenticado.

Parámetros

- **vehiculo_id** (**int**) – ID del vehículo para el que se desean consultar los errores.
- **usuario** (**Usuario**) – Usuario autenticado mediante JWT.
- **db** (**Session**) – Sesión activa de la base de datos.

Devuelve
Lista de errores registrados.

Tipo del valor devuelto
List[*ErrorVehiculo*]

Muestra
HTTPException 404 – Si no existen errores para ese vehículo.

main.obtener_usuario_desde_token(*token: str = Depends(OAuth2PasswordBearer)*, *db: Session = Depends(get_db)*)

Extrae y valida el usuario actual a partir del token JWT proporcionado.

Parámetros

- **token** (**str**) – Token JWT incluido en el encabezado de autorización.
- **db** (**Session**) – Sesión de base de datos.

Devuelve
Instancia del usuario autenticado.

Tipo del valor devuelto
Usuario

Muestra
HTTPException 401 – Si el token es inválido o ha expirado.

main.obtener_vehiculo(*vehiculo_id: int*, *usuario:* Usuario = *Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*)

Recupera la información de un vehículo específico registrado por el usuario autenticado.

Parámetros

- `vehiculo_id` (`int`) – ID del vehículo a consultar.
- `usuario` (`Usuario`) – Usuario autenticado mediante JWT.
- `db` (`Session`) – Sesión activa de la base de datos.

Devuelve
Objeto del vehículo solicitado.

Tipo del valor devuelto
Vehiculo

Muestra
`HTTPException 404` – Si el vehículo no pertenece al usuario o no existe.

`main.obtener_vehiculos(`*usuario:* Usuario *= Depends(obtener_usuario_desde_token)*, *db: Session = Depends(get_db)*`)`

Obtiene todos los vehículos registrados por el usuario autenticado.

Parámetros

- `db` (`Session`) – Sesión de base de datos.
- `usuario` (`Usuario`) – Usuario autenticado mediante JWT.

Devuelve
Lista de vehículos asociados al usuario.

Tipo del valor devuelto
List[VehiculoBase]

`main.register(`*datos:* UsuarioRegistro, *db: Session = Depends(get_db)*`)`

POST `/register`

Registra un nuevo usuario en la base de datos.

Parámetros: - `datos` (UsuarioRegistro): Objeto que contiene el nombre de usuario y la contraseña. - `db` (Session): Sesión activa de la base de datos, proporcionada por FastAPI.

Retorna: - `dict`: Un mensaje indicando si el usuario fue registrado exitosamente.

Errores: - `400`: Si los campos son inválidos o el nombre de usuario ya existe.

`async main.saludo()`

Devuelve un mensaje simple para verificar que la API está activa.

Este endpoint puede utilizarse para pruebas de conectividad o para confirmar que el backend está desplegado correctamente.

Devuelve
Mensaje de saludo indicando que la API funciona.

Tipo del valor devuelto
dict

main.ver_informe(*token: str*, *db: Session = Depends(get_db)*)

Devuelve los datos del informe generado a partir de un token único.

Este endpoint permite el acceso público a un informe de diagnóstico de vehículo mediante un enlace con token generado previamente. No requiere autenticación, pero valida que el token sea legítimo.

Parámetros
token (str) – Token único del informe generado.

Devuelve
Información del vehículo (marca, modelo, año, etc.) y lista de errores DTC.

Tipo del valor devuelto
dict

Muestra

- HTTPException 400 – Si el token no es válido o demasiado corto.
- HTTPException 404 – Si no se encuentra el informe, el vehículo o los errores asociados.
- HTTPException 500 – Si ocurre un error inesperado al procesar la solicitud.

main.verificar_password(*plain_password*, *hashed_password*)

Verifica si una contraseña en texto plano coincide con su hash almacenado.

Parámetros

- password_plano (str) – Contraseña proporcionada por el usuario.
- password_hash (str) – Hash almacenado en la base de datos.

Devuelve
True si coinciden, False si no.

Tipo del valor devuelto
bool

Capítulo 4

Configuración del Proyecto

En esta sección se describe la configuración general del proyecto, desde la conexión a la base de datos hasta los parámetros de correo y seguridad. Toda la configuración se basa en variables de entorno que puedes definir en un archivo *.env* o directamente en tu entorno.

4.1. Configuración de Entorno

Para que la aplicación funcione correctamente, debes definir las siguientes variables de entorno (pueden estar en un archivo *.env*):

Variable	Descripción
`DATABASE_URL`	URL de conexión a la base de datos MySQL (p. ej.: `mysql+pymysql://user:password@host:port/dbname`).
`MAIL_USERNAME`	Usuario para SMTP (FastAPI-Mail).
`MAIL_PASSWORD`	Contraseña para SMTP.
`MAIL_FROM`	Correo desde el cual se enviarán los mensajes.
`MAIL_PORT`	Puerto SMTP (por defecto: 587).
`MAIL_SERVER`	Servidor SMTP (p. ej.: `smtp.gmail.com`).
`MAIL_STARTTLS`	`True` si se debe usar STARTTLS.
`MAIL_SSL_TLS`	`True` si se debe usar SSL/TLS.
`USE_CREDENTIALS`	`True` si se usan las credenciales definidas.
`VALIDATE_CERTS`	`True` para validar certificados SSL.
`SECRET_KEY`	Clave secreta para firmar JWT.
`ALGORITHM`	Algoritmo JWT (p. ej.: `HS256`).
`ACCESS_TOKEN_EXPIRE_MINUTES`	Tiempo de expiración del token (en minutos).

4.2. Base de Datos

- **URL**: obtenida desde `DATABASE_URL`.
- **Motor**: MySQL con el driver PyMySQL.
- **ORM**: SQLAlchemy.
- **Sesiones**: se genera con `SessionLocal()` por request.

Declaración en código (`main.py`):

```
# Configuración de la base de datos
DATABASE_URL = os.getenv("DATABASE_URL", "mysql+pymysql://user:password@db/
↪talleres")
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

4.3. Correo Electrónico

La aplicación utiliza **FastAPI-Mail** para el envío de correos electrónicos (por ejemplo, para enviar informes generados).

- Se configuran las credenciales y parámetros de SMTP en la clase `ConnectionConfig`.
- Se soporta TLS y SSL según las variables `MAIL_STARTTLS` y `MAIL_SSL_TLS`.
- Ejemplo de uso en `main.py`:

```
conf = ConnectionConfig(
    MAIL_USERNAME=os.getenv("MAIL_USERNAME"),
    MAIL_PASSWORD=os.getenv("MAIL_PASSWORD"),
    MAIL_FROM=os.getenv("MAIL_FROM"),
    MAIL_PORT=int(os.getenv("MAIL_PORT", 587)),
    MAIL_SERVER=os.getenv("MAIL_SERVER"),
    MAIL_STARTTLS=os.getenv("MAIL_STARTTLS", "True") == "True",
    MAIL_SSL_TLS=os.getenv("MAIL_SSL_TLS", "False") == "True",
    USE_CREDENTIALS=os.getenv("USE_CREDENTIALS", "True") == "True",
    VALIDATE_CERTS=os.getenv("VALIDATE_CERTS", "True") == "True"
)
fm = FastMail(conf)
```

4.4. Seguridad

La autenticación y autorización se basan en JWT. A continuación se detallan los componentes clave:

- **SECRET_KEY**: clave secreta para firmar y verificar tokens JWT. Defínela en la variable de entorno `SECRET_KEY` o déjala en el código como fallback.
- **ALGORITHM**: algoritmo para JWT, por defecto `HS256`.
- **ACCESS_TOKEN_EXPIRE_MINUTES**: tiempo de expiración del token (en minutos). Se obtiene de la variable de entorno `ACCESS_TOKEN_EXPIRE_MINUTES` o usa 300 por defecto.
- Para hashear contraseñas se utiliza `bcrypt` a través de `passlib`:

  ```
  pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
  ```

- Para extraer el token de las peticiones, se usa `OAuth2PasswordBearer(tokenUrl="login")`.

Flujo de autenticación en `main.py`:

1. **Registro**: el endpoint `/register` recibe un `UsuarioRegistro` (username + password), hashea la contraseña y la almacena.
2. **Login**: el endpoint `/login` recibe un `UsuarioLogin`, verifica credenciales y, si es válido, genera un JWT con `crear_token`.
3. **Dependencia ``obtener_usuario_desde_token``**: se encarga de decodificar el JWT, extraer el `sub` (username) y cargar el usuario desde la base de datos. Si no encuentra usuario o el token está expirado, lanza `HTTPException(401)`.

4.5. Dependencias reutilizables

A continuación se listan las funciones que puedes reusar en tus endpoints:

`main.get_db()`

Dependencia de FastAPI para obtener una sesión de base de datos.

Se utiliza con *Depends(get_db)* para abrir una sesión, cederla al endpoint y cerrarla automáticamente.

`main.obtener_usuario_desde_token(`*token: str = Depends(OAuth2PasswordBearer)*, *db: Session = Depends(get_db)*`)`

Extrae y valida el usuario actual a partir del token JWT proporcionado.

Parámetros

- `token` (`str`) – Token JWT incluido en el encabezado de autorización.
- `db` (`Session`) – Sesión de base de datos.

Devuelve

Instancia del usuario autenticado.

Tipo del valor devuelto

Usuario

Muestra

`HTTPException 401` – Si el token es inválido o ha expirado.

- `get_db()`: - Abre una sesión de base de datos (SQLAlchemy) y la cierra automáticamente al salir del contexto. - Uso típico:

```
@app.get("/ruta_ejemplo")
def ejemplo(db: Session = Depends(get_db)):
    # db es una sesión válida
    ...
```

- `obtener_usuario_desde_token`: - Extrae el token JWT del encabezado `Authorization` y devuelve la instancia de `Usuario` correspondiente. - Uso típico:

```
@app.get("/ruta_protegida")
def protegida(usuario: Usuario = Depends(obtener_usuario_desde_
↪token)):
    # usuario es el objeto Usuario autenticado
    ...
```

—

Esta documentación está estructurada de la siguiente manera:

- **Introducción**: visión general del proyecto, tecnologías utilizadas y objetivos.
- **Endpoints**: listado completo de rutas disponibles, con ejemplos, parámetros y respuestas.
- **Modelos de Datos**: descripción de las entidades ORM (SQLAlchemy) y los esquemas de validación (Pydantic).
- **Configuración**: detalles sobre variables de entorno, conexión a bases de datos, correo electrónico, JWT y seguridad.

Tabla de enrutamiento HTTP

Índice de Módulos Python

Índice

S

T

U

V

Y