# Proyecto: ScooterFlow API

## Introducción

## ¿Qué es DevOps y cómo lo aplicamos en ScooterFlow?

**DevOps** es un conjunto de prácticas, herramientas y una filosofía cultural que une a los equipos de desarrollo (**Dev**) y operaciones de TI (**Ops**) para automatizar e integrar sus procesos. Su objetivo es acortar el ciclo de desarrollo, aumentando la velocidad, calidad y confiabilidad en la entrega de software.

En este proyecto vas a actuar como un **Ingeniero DevOps** aplicando estos tres pilares:

### 1. Infraestructura como Código (IaC) con Docker

En lugar de instalar bases de datos manualmente, definimos nuestra infraestructura en archivos (docker-compose.yml). Esto garantiza que el entorno de desarrollo sea **idéntico** al de producción, eliminando el clásico "en mi ordenador funcionaba".

### 2. Integración Continua (CI) con GitHub Actions

La "C" de CI/CD consiste en automatizar las pruebas. Cada vez que subas código, un robot (GitHub Actions) ejecutará tus tests de pytest. Si algo falla, el sistema te avisa antes de que el error llegue al usuario final. Esto asegura la **calidad y confiabilidad**.

### 3. Gestión de la Evolución con Alembic

DevOps también trata sobre cómo gestionamos los cambios en los datos. Con **Alembic**, tratamos las modificaciones de la base de datos como si fueran versiones de código, permitiendo evolucionar la API de patinetes sin romper la integridad de la información existente.

# Escenario

La startup **ScooterFlow** te pide diseñar el núcleo de su sistema de gestión de micro-movilidad. El objetivo es construir una API robusta que gestione una flota de patinetes eléctricos y las zonas de la ciudad donde operan.

# Stack Tecnológico Obligatorio

- **Framework:** FastAPI.
- **Base de Datos:** PostgreSQL.
- **ORM:** SQLAlchemy.
- **Migraciones:** Alembic.
- **Contenedores:** Docker & Docker Compose.
- **CI/CD:** GitHub Actions.

---

# Requisitos del Sistema

## 1. Modelos de Datos (Relación 1:N)

Debes implementar dos tablas relacionadas:

1. **Zona (Zone):** `id`, `nombre` (ej: "Centro Histórico"), `codigo_postal`, `limite_velocidad`.
2. **Patinete (Scooter):** `id`, `numero_serie`, `modelo`, `bateria` (0-100), `estado`, `zona_id` (Relación con Zona).

## 2. Estados del Patinete

El campo `estado` debe ser un **Enum** de Python con los siguientes valores:

- `disponible`, `en_uso`, `mantenimiento`, `sin_bateria`.

---

# Tareas a Realizar

## Tarea 1: Infraestructura Docker

Crea un entorno con **Docker Compose** que levante dos servicios:

- `db`: Imagen de PostgreSQL 15-alpine.
- `api`: Tu aplicación FastAPI construida mediante un `Dockerfile`.

- **Reto:** La API debe esperar a que la DB esté lista antes de intentar conectar.

### Tarea 2: Evolución con Alembic (Migraciones)

No puedes crear las tablas con `Base.metadata.create_all()`. Debes usar Alembic:

1. Genera una migración inicial con las tablas base.
2. **Segunda migración:** Añade un campo `puntuacion_usuario` (float) a la tabla Patinete una vez que la base ya esté creada.

### Tarea 3: Lógica de Negocio y Validaciones

- **Validación Pydantic:** Asegura que nadie pueda crear un patinete con batería menor a 0 o mayor a 100.
- **Endpoint Especial:** Crea una ruta `POST /zonas/{zona_id}/mantenimiento`. Al llamarla, todos los patinetes de esa zona con batería **inferior al 15%** deben cambiar automáticamente su estado a `mantenimiento`.

### Tarea 4: Batería de Tests

Escribe al menos 5 tests usando `pytest`:

- Verificar que un patinete se crea correctamente vinculado a una zona.
- Verificar que la validación de batería funciona (error 422 si pongo 150%).
- Verificar que la lógica de "paso a mantenimiento" funciona correctamente.

### Tarea 5: CI/CD en GitHub

Configura un Workflow de **GitHub Actions** que:

1. Levante una base de datos Postgres de prueba.
2. Instale las dependencias.
3. Ejecute los tests en cada `push` a la rama `main`.
- **Nota:** No se evaluarán proyectos con el "check" de GitHub en **rojo**.

---

## Criterios de Entrega

1. Enlace al repositorio de GitHub (público).

2. El README del repositorio debe explicar cómo levantar el proyecto en un solo comando.
3. Incluir el archivo de historial de migraciones de Alembic.

---

## Pistas

- No olvides configurar el PYTHONPATH en tu archivo de CI para que encuentre la carpeta app.
- Revisa la documentación de SQLAlchemy para las ForeignKey y relationship.
- En Alembic, recuerda importar tus modelos en env.py para que autogenerate funcione.

---

## **Ayuda:** Guia paso a paso de una primera migración con Alembic:

Para que Alembic funcione, debemos seguir un orden lógico. Si saltas un paso, el "cableado" no funcionará.

### Paso 1: Inicialización

Dentro de la carpeta de tu proyecto (y con el contenedor de la API corriendo), ejecuta:

Bash

```
docker-compose exec api alembic init migrations
```

*Esto creará una carpeta llamada migrations y un archivo alembic.ini.*

### Paso 2: Conectar Alembic con tus Modelos (El paso crítico)

Alembic no sabe qué has programado en models.py. Tienes que "presentárselos". Edita el archivo **migrations/env.py**:

Python

```
# 1. Importa la Base y tus modelos
from app.database import Base
from app.models import Scooter, Zone

# 2. Busca la línea que dice target_metadata = None y cámbiala:
```

```
target_metadata = Base.metadata
```

### Paso 3: Configurar la URL de la Base de Datos

Alembic necesita saber dónde está PostgreSQL:

Python
```
import os
# Al principio del archivo env.py:
config.set_main_option("sqlalchemy.url", os.getenv("DATABASE_URL"))
```

### Paso 4: Crear la "Revisión" (El borrador)

Ahora le pedimos a Alembic que compare tus modelos con la base de datos vacía:

Bash
```
docker-compose exec api alembic revision --autogenerate -m "Crear tablas iniciales"
```

*Si todo va bien, se creará un archivo dentro de migrations/versions/ con el código Python para crear las tablas.*

### Paso 5: Aplicar los cambios (El momento de la verdad)

Hasta ahora la base de datos sigue vacía. Para ejecutar el código del paso anterior:

Bash
```
docker-compose exec api alembic upgrade head
```

---

## ¿Cómo añadir un campo nuevo después? (Ej: puntuacion)

Si a mitad del proyecto decides añadir un campo, el flujo es:

1. **Modificas** tu clase en models.py (añades el campo).
2. **Generas** una nueva revisión: docker-compose exec api alembic revision --autogenerate -m "añadir puntuacion"
3. **Aplicas**: docker-compose exec api alembic upgrade head

---

### ¿Por qué podría fallar Alembic? (Checklist de errores)

- **Error de conexión:** ¿Has puesto `db` como host en la URL en lugar de `localhost`?
- **No detecta cambios:** ¿Has olvidado importar los modelos en `env.py`?
- **Tabla `alembic_version` bloqueada:** A veces, si una migración falla a mitad, la base de datos queda "esperando". Reinicia los contenedores.

## Ayuda para la Tarea 1: Controlar el arranque de la DB

Para cumplir el "Reto" de que la API espere a la base de datos, lo más sencillo hoy en día es usar el **Healthcheck** nativo de Docker. Sugiero que el archivo `docker-compose.yml` se parezca a esto:

YAML

```yaml
services:
  db:
    image: postgres:15-alpine
    # ... resto de la configuración ...
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U salva -d productos"]
      interval: 5s
      timeout: 5s
      retries: 5

  api:
    build: .
    # ... resto de la configuración ...
    depends_on:
      db:
        condition: service_healthy  # <--- Esto hace la magia
```