Guía Avanzada: Backend de Restaurante con FastAPI + MySQL

Fecha: 21/08/2025
Proyecto: API de gestión de Platos y Vinos

Resumen

Esta guía detalla la arquitectura, el modelo de datos y la implementación de un backend para una carta de restaurante basado en FastAPI y MySQL. Está adaptada a tu README y amplía con recomendaciones prácticas, ejemplos de código completos, pautas de migración, pruebas y despliegue.

Índice

1. Objetivos y requisitos no funcionales

- Mantener separación de capas: rutas ↔ schemas ↔ modelos ↔ base de datos.
- Creación automática de tablas en desarrollo con SQLAlchemy.create_all() (sin borrar/alterar existentes).
- Evolución a migraciones con Alembic para cambios estructurales en producción.
- Código legible, testeable y con convenciones (black/isort/pytest).
- Facilidad para añadir autenticación y autorización más adelante.

2. Modelo de datos actual y mejoras propuestas

Según tu README, Platos y Vinos tienen categorías y, en el caso de Vinos, varios campos opcionales. Para evitar problemas con búsquedas y evolución, se recomienda normalizar:

- Separar categorías en tablas dedicadas: plato_categorias y vino_categorias.
- Modelar alérgenos como una tabla con relación many-to-many con platos.
- Modelar tipos de uva como many-to-many con vinos.
- Añadir índices en campos consultados (nombre, categoría, precio).

```
-- Opción normalizada (recomendada)
platos(id PK, nombre, precio, descripcion?, categoria_id FK)
plato_categorias(id PK, nombre UNIQUE)

alergenos(id PK, nombre UNIQUE)
platos_alergenos(plato_id FK, alergeno_id FK, PK compuesta)

vinos(id PK, nombre, precio, bodega?, denominacion?, enologo?, categoria_id FK)
vino_categorias(id PK, nombre UNIQUE)

uvas(id PK, nombre UNIQUE)
vinos_uvas(vino_id FK, uva_id FK, PK compuesta)
```

3. Estructura del proyecto (adaptada a /src)

```
Fast-Api-Backend/
 └── src/
    ├── auth/                # (futuro: JWT/API Key)
    ├── database/              # engine, SessionLocal, Base, init_db, get_db
    │   └── __init__.py
    ├── entities/             # Modelos SQLAlchemy
    │   ├── plato.py
    │   ├── vino.py
    │   ├── categoria.py         # categorías plato/vino
    │   ├── alergeno.py          # + tabla asociación platos_alergenos
    │   └── uva.py              # + tabla asociación vinos_uvas
    ├── schemas/               # Esquemas Pydantic
    │   ├── plato.py
    │   ├── vino.py
    │   └── comunes.py
    ├── routes/               # Routers por recurso
    │   ├── platos.py
    │   └── vinos.py
    ├── main.py
    └── __init__.py
```

4. Configuración: settings, .env, base de datos y create_all()

```
# .env (desarrollo local con Docker ejecutando MySQL en puerto 9000)
APP_NAME="Restaurante API"
```

```
ENV="dev"
DB_USER="root"
DB_PASSWORD="123456789"
DB_HOST="127.0.0.1"
DB_PORT=9000
DB_NAME="restaurantdb"
SQL_ECHO=false
# src/database/__init__.py
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base
import os

APP_NAME = os.getenv("APP_NAME", "Restaurante API")
ENV = os.getenv("ENV", "dev")
DB_USER = os.getenv("DB_USER", "root")
DB_PASSWORD = os.getenv("DB_PASSWORD", "123456789")
DB_HOST = os.getenv("DB_HOST", "127.0.0.1")
DB_PORT = int(os.getenv("DB_PORT", "9000"))  # mapeado a 3306 en Docker
DB_NAME = os.getenv("DB_NAME", "restaurantdb")
SQL_ECHO = os.getenv("SQL_ECHO", "false").lower() == "true"

DATABASE_URI = f"mysql+pymysql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}?charset=ut

engine = create_engine(DATABASE_URI, echo=SQL_ECHO, pool_pre_ping=True, pool_recycle=3600, future=True)
SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False, future=True)
Base = declarative_base()

def init_db():
    # Importa todos los modelos para que estén en Base.metadata
    from src.entities import plato, vino, categoria, alergeno, uva  # noqa: F401
    Base.metadata.create_all(bind=engine)  # crea si no existen (no borra)

# Dependencia FastAPI
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

5. Modelos SQLAlchemy (normalizados)

```
# src/entities/categoria.py
from sqlalchemy import Column, Integer, String
from src.database import Base

class PlatoCategoria(Base):
    __tablename__ = "plato_categorias"
    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(50), unique=True, nullable=False)

class VinoCategoria(Base):
    __tablename__ = "vino_categorias"
    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(50), unique=True, nullable=False)
# src/entities/alergeno.py
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship
from src.database import Base
platos_alergenos = Table(
```

```
    "platos_alergenos",
    Base.metadata,
    Column("plato_id", ForeignKey("platos.id"), primary_key=True),
    Column("alergeno_id", ForeignKey("alergenos.id"), primary_key=True),
)

class Alergeno(Base):
    __tablename__ = "alergenos"
    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(100), unique=True, nullable=False)
# src/entities/uva.py
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship
from src.database import Base

vinos_uvas = Table(
    "vinos_uvas",
    Base.metadata,
    Column("vino_id", ForeignKey("vinos.id"), primary_key=True),
    Column("uva_id", ForeignKey("uvas.id"), primary_key=True),
)

class Uva(Base):
    __tablename__ = "uvas"
    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(100), unique=True, nullable=False)
# src/entities/plato.py
from sqlalchemy import Column, Integer, String, Float, Text, ForeignKey
from sqlalchemy.orm import relationship
from src.database import Base
from src.entities.alergeno import platos_alergenos

class Plato(Base):
    __tablename__ = "platos"

    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(100), nullable=False, index=True)
    precio = Column(Float, nullable=False)
    descripcion = Column(Text, nullable=True)
    categoria_id = Column(Integer, ForeignKey("plato_categorias.id"), nullable=False)

    # relación many-to-many con alergenos
    alergenos = relationship("Alergeno", secondary=platos_alergenos, backref="platos")
# src/entities/vino.py
from sqlalchemy import Column, Integer, String, Float, ForeignKey
from sqlalchemy.orm import relationship
from src.database import Base
from src.entities.uva import vinos_uvas

class Vino(Base):
    __tablename__ = "vinos"

    id = Column(Integer, primary_key=True, index=True)
    nombre = Column(String(100), nullable=False, index=True)
    precio = Column(Float, nullable=False)
    bodega = Column(String(100), nullable=True)
    denominacion = Column(String(100), nullable=True)
    enologo = Column(String(100), nullable=True)
    categoria_id = Column(Integer, ForeignKey("vino_categorias.id"), nullable=False)
```

```
    # relación many-to-many con uvas
    uvas = relationship("Uva", secondary=vinos_uvas, backref="vinos")
```

6. Schemas Pydantic (validación y serialización)

```
# src/schemas/plato.py
from pydantic import BaseModel, Field
from typing import Optional, List

class PlatoBase(BaseModel):
    nombre: str = Field(min_length=1, max_length=100)
    precio: float = Field(gt=0)
    descripcion: Optional[str] = None
    categoria_id: int
    alergenos_ids: List[int] = []

class PlatoCreate(PlatoBase):
    pass

class PlatoUpdate(BaseModel):
    nombre: Optional[str] = Field(default=None, min_length=1, max_length=100)
    precio: Optional[float] = Field(default=None, gt=0)
    descripcion: Optional[str] = None
    categoria_id: Optional[int] = None
    alergenos_ids: Optional[List[int]] = None

class PlatoOut(BaseModel):
    id: int
    nombre: str
    precio: float
    descripcion: Optional[str]
    categoria_id: int
    alergenos: List[str] = []

    class Config:
        from_attributes = True
# src/schemas/vino.py
from pydantic import BaseModel, Field
from typing import Optional, List

class VinoBase(BaseModel):
    nombre: str = Field(min_length=1, max_length=100)
    precio: float = Field(gt=0)
    bodega: Optional[str] = None
    denominacion: Optional[str] = None
    enologo: Optional[str] = None
    categoria_id: int
    uvas_ids: List[int] = []

class VinoCreate(VinoBase):
    pass

class VinoUpdate(BaseModel):
    nombre: Optional[str] = Field(default=None, min_length=1, max_length=100)
    precio: Optional[float] = Field(default=None, gt=0)
    bodega: Optional[str] = None
    denominacion: Optional[str] = None
    enologo: Optional[str] = None
    categoria_id: Optional[int] = None
    uvas_ids: Optional[List[int]] = None
```

```
class VinoOut(BaseModel):
    id: int
    nombre: str
    precio: float
    bodega: Optional[str]
    denominacion: Optional[str]
    enologo: Optional[str]
    categoria_id: int
    uvas: List[str] = []

    class Config:
        from_attributes = True
```

7. Endpoints FastAPI (CRUD, filtros, paginación)

```
# src/routes/platos.py
from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlalchemy.orm import Session
from typing import List, Optional

from src.database import get_db
from src.entities.plato import Plato
from src.entities.alergeno import Alergeno
from src.schemas.plato import PlatoCreate, PlatoUpdate, PlatoOut

router = APIRouter(prefix="/platos", tags=["platos"])

@router.get("", response_model=List[PlatoOut])
def listar_platos(
    db: Session = Depends(get_db),
    categoria_id: Optional[int] = None,
    q: Optional[str] = None,
    limit: int = Query(50, ge=1, le=200),
    offset: int = Query(0, ge=0),
    ordenar_por: str = Query("nombre")  # nombre|precio
):
    query = db.query(Plato)
    if categoria_id is not None:
        query = query.filter(Plato.categoria_id == categoria_id)
    if q:
        like = f"%{q}%"
        query = query.filter(Plato.nombre.like(like))
    if ordenar_por == "precio":
        query = query.order_by(Plato.precio.asc())
    else:
        query = query.order_by(Plato.nombre.asc())
    return query.offset(offset).limit(limit).all()

@router.post("", response_model=PlatoOut, status_code=status.HTTP_201_CREATED)
def crear_plato(payload: PlatoCreate, db: Session = Depends(get_db)):
    plato = Plato(
        nombre=payload.nombre,
        precio=payload.precio,
        descripcion=payload.descripcion,
        categoria_id=payload.categoria_id,
    )
    if payload.alergenos_ids:
        alergenos = db.query(Alergeno).filter(Alergeno.id.in_(payload.alergenos_ids)).all()
        plato.alergenos = alergenos
    db.add(plato)
```

```
    db.commit()
    db.refresh(plato)
    return plato

@router.get("/{plato_id}", response_model=PlatoOut)
def obtener_plato(plato_id: int, db: Session = Depends(get_db)):
    plato = db.query(Plato).get(plato_id)
    if not plato:
        raise HTTPException(status_code=404, detail="Plato no encontrado")
    return plato

@router.patch("/{plato_id}", response_model=PlatoOut)
def actualizar_plato(plato_id: int, payload: PlatoUpdate, db: Session = Depends(get_db)):
    plato = db.query(Plato).get(plato_id)
    if not plato:
        raise HTTPException(status_code=404, detail="Plato no encontrado")
    data = payload.dict(exclude_unset=True)
    alergenos_ids = data.pop("alergenos_ids", None)
    for k, v in data.items():
        setattr(plato, k, v)
    if alergenos_ids is not None:
        alergenos = db.query(Alergeno).filter(Alergeno.id.in_(alergenos_ids)).all()
        plato.alergenos = alergenos
    db.commit()
    db.refresh(plato)
    return plato

@router.delete("/{plato_id}", status_code=status.HTTP_204_NO_CONTENT)
def borrar_plato(plato_id: int, db: Session = Depends(get_db)):
    plato = db.query(Plato).get(plato_id)
    if not plato:
        raise HTTPException(status_code=404, detail="Plato no encontrado")
    db.delete(plato)
    db.commit()
    return None
# src/routes/vinos.py
from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlalchemy.orm import Session
from typing import List, Optional

from src.database import get_db
from src.entities.vino import Vino
from src.entities.uva import Uva
from src.schemas.vino import VinoCreate, VinoUpdate, VinoOut

router = APIRouter(prefix="/vinos", tags=["vinos"])

@router.get("", response_model=List[VinoOut])
def listar_vinos(
    db: Session = Depends(get_db),
    categoria_id: Optional[int] = None,
    q: Optional[str] = None,
    limit: int = Query(50, ge=1, le=200),
    offset: int = Query(0, ge=0),
    ordenar_por: str = Query("nombre")  # nombre|precio
):
    query = db.query(Vino)
    if categoria_id is not None:
        query = query.filter(Vino.categoria_id == categoria_id)
```

```
    if q:
        like = f"%{q}%"
        query = query.filter(Vino.nombre.like(like))
    if ordenar_por == "precio":
        query = query.order_by(Vino.precio.asc())
    else:
        query = query.order_by(Vino.nombre.asc())
    return query.offset(offset).limit(limit).all()

@router.post("", response_model=VinoOut, status_code=status.HTTP_201_CREATED)
def crear_vino(payload: VinoCreate, db: Session = Depends(get_db)):
    vino = Vino(
        nombre=payload.nombre,
        precio=payload.precio,
        bodega=payload.bodega,
        denominacion=payload.denominacion,
        enologo=payload.enologo,
        categoria_id=payload.categoria_id,
    )
    if payload.uvas_ids:
        uvas = db.query(Uva).filter(Uva.id.in_(payload.uvas_ids)).all()
        vino.uvas = uvas
    db.add(vino)
    db.commit()
    db.refresh(vino)
    return vino

@router.get("/{vino_id}", response_model=VinoOut)
def obtener_vino(vino_id: int, db: Session = Depends(get_db)):
    vino = db.query(Vino).get(vino_id)
    if not vino:
        raise HTTPException(status_code=404, detail="Vino no encontrado")
    return vino

@router.patch("/{vino_id}", response_model=VinoOut)
def actualizar_vino(vino_id: int, payload: VinoUpdate, db: Session = Depends(get_db)):
    vino = db.query(Vino).get(vino_id)
    if not vino:
        raise HTTPException(status_code=404, detail="Vino no encontrado")
    data = payload.dict(exclude_unset=True)
    uvas_ids = data.pop("uvas_ids", None)
    for k, v in data.items():
        setattr(vino, k, v)
    if uvas_ids is not None:
        uvas = db.query(Uva).filter(Uva.id.in_(uvas_ids)).all()
        vino.uvas = uvas
    db.commit()
    db.refresh(vino)
    return vino

@router.delete("/{vino_id}", status_code=status.HTTP_204_NO_CONTENT)
def borrar_vino(vino_id: int, db: Session = Depends(get_db)):
    vino = db.query(Vino).get(vino_id)
    if not vino:
        raise HTTPException(status_code=404, detail="Vino no encontrado")
    db.delete(vino)
    db.commit()
    return None
```

8. main.py, CORS y versionado

```
# src/main.py
import os
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from src.database import init_db
from src.routes.platos import router as platos_router
from src.routes.vinos import router as vinos_router

APP_NAME = os.getenv("APP_NAME", "Restaurante API")
ENV = os.getenv("ENV", "dev")

app = FastAPI(title=APP_NAME, version="1.0.0")

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173", "http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.on_event("startup")
def startup_event():
    if ENV == "dev":
        init_db()  # crea tablas si no existen

app.include_router(platos_router, prefix="/api/v1")
app.include_router(vinos_router, prefix="/api/v1")
```

9. Docker & docker-compose (MySQL en :9000)

```
# docker run (rápido)
# docker run --name restaurantdb -e MYSQL_ROOT_PASSWORD=123456789 -p 9000:3306 -d mysql:8.0

# docker-compose.yml (recomendado)
version: "3.9"
services:
  db:
    image: mysql:8.0
    container_name: restaurantdb
    environment:
      MYSQL_ROOT_PASSWORD: 123456789
      MYSQL_DATABASE: restaurantdb
    ports:
      - "9000:3306"
    command: ["--default-authentication-plugin=mysql_native_password",
              "--character-set-server=utf8mb4",
              "--collation-server=utf8mb4_unicode_ci"]
    volumes:
      - db_data:/var/lib/mysql

  api:
    build: .
    command: uvicorn src.main:app --host 0.0.0.0 --port 8000 --reload
    environment:
      APP_NAME: "Restaurante API"
      ENV: "dev"
```

```
      DB_USER: "root"
      DB_PASSWORD: "123456789"
      DB_HOST: "db"
      DB_PORT: 3306
      DB_NAME: "restaurantdb"
      SQL_ECHO: "false"
    volumes:
      - .:/code
    working_dir: /code
    depends_on:
      - db
    ports:
      - "8000:8000"

volumes:
  db_data:
```

10. Pruebas con pytest (unitarias y API)

```
# tests/test_api.py
from fastapi.testclient import TestClient
from src.main import app

client = TestClient(app)

def test_docs_up():
    r = client.get("/docs")
    assert r.status_code == 200

def test_listar_platos():
    r = client.get("/api/v1/platos")
    assert r.status_code == 200
    assert isinstance(r.json(), list)
```

11. Migraciones con Alembic (camino recomendado a prod)

```
pip install alembic
alembic init alembic

# En alembic/env.py:
# from src.database import Base, DATABASE_URI
# target_metadata = Base.metadata
# config.set_main_option("sqlalchemy.url", DATABASE_URI)

# Crear migración y aplicar:
alembic revision --autogenerate -m "init schema"
alembic upgrade head
```

12. Seguridad básica (auth opcional), validación, errores

- Valida datos con Pydantic (longitudes, rangos de precio).
- Limita orígenes CORS a tu frontend.
- Añade auth simple (API Key) o JWT cuando publiques.
- Maneja errores con HTTPException y responses coherentes.

13. Logging, observabilidad y rendimiento

- Activa SQLAlchemy echo solo en dev; en prod usa logging INFO.
- Índices en nombre/categoría para filtros; considera full-text en MySQL si buscas por descripción.

- Paginación por defecto (limit/offset) para endpoints de listado.

14. Seed de datos (categorías iniciales)

```
# scripts/seed.py
from src.database import SessionLocal, init_db
from src.entities.categoria import PlatoCategoria, VinoCategoria

def run():
    init_db()
    db = SessionLocal()
    try:
        base_platos = ["Entrantes", "Platos principales", "Postres"]
        base_vinos = ["Vinos Blancos", "Vinos Tintos", "Vinos Dulces"]
        for nombre in base_platos:
            if not db.query(PlatoCategoria).filter_by(nombre=nombre).first():
                db.add(PlatoCategoria(nombre=nombre))
        for nombre in base_vinos:
            if not db.query(VinoCategoria).filter_by(nombre=nombre).first():
                db.add(VinoCategoria(nombre=nombre))
        db.commit()
    finally:
        db.close()

if __name__ == "__main__":
    run()
```

15. Checklist de despliegue y Roadmap

- Variables de entorno en prod (sin credenciales en repo).
- Migraciones Alembic aplicadas antes del arranque.
- Healthcheck, logs centralizados, backups de MySQL.

Roadmap sugerido:

- v0: CRUD básico y seed categorías.
- v1: Alembic + auth básica (API Key/JWT) + CI.
- v2: Búsqueda avanzada, full-text, cache selectiva.

16. Recomendaciones y adaptaciones específicas para tu repo

- Mantener /src como raíz de código y añadir carpeta /schemas y /scripts.
- Cambiar campos 'alérgenos' y 'tipos de uva' de texto plano a tablas relacionadas para filtros robustos.
- Usar create_all() solo en dev; planificar migraciones Alembic para cada cambio de esquema.
- Agregar validaciones de rango a precios y longitudes de nombre/descripcion.
- Crear endpoints de lectura de categorías (GET /categorias/platos, /categorias/vinos).