Guía de arquitectura y mejoras (FastAPI + SQLAlchemy + Pydantic v2)

Te dejo una guía práctica, paso a paso, con **qué cambiar**, **por qué** y **cómo** hacerlo para que tu backend sea mantenible a largo plazo. Está basada en tu documentación y ejemplos (esquema, carpetas, endpoints y configuración) y la actualiza con **mejores prácticas modernas** (SQLAlchemy 2.0, Pydantic v2, validación estricta, migraciones, testing y despliegue).

---

1) Visión general: prioridades

1. **Modelo de datos saneado y estable** (índices correctos, DECIMAL para dinero, M:N bien modeladas).
2. **Configuración limpia** (settings/.env, create_all solo en dev, migraciones en prod).
3. **Capa web delgada** (routers) + **servicios** (negocio) + **repositorios** (persistencia) para testear fácil.
4. **Pydantic v2** con tipos estrictos y `from_attributes=True` (antes `orm_mode`).
5. **Errores, paginación, filtros y seguridad** (CORS correcto y JWT cuando publiques).

---

2) Estructura de carpetas (organización por capas)

```
src/
├─ core/               # config, logging, seguridad, errores
├─ database/           # engine, sesiones, base, init_db
├─ entities/           # modelos SQLAlchemy 2.0 (tipados)
├─ repositories/       # consultas CRUD por agregado (platos, vinos...)
├─ services/           # lógica de negocio (orquestación)
├─ schemas/            # Pydantic v2 (request/response)
├─ api/
│  ├─ v1/
│  │  ├─ platos.py
│  │  └─ vinos.py
│  └─ deps.py          # dependencias (get_db, seguridad...)
├─ main.py
└─ scripts/            # seed, tareas admin
```

Por qué: los routers solo coordinan; el negocio vive en `services/` y el acceso a datos en `repositories/`. Así reduces acoplamiento y los tests son triviales.

---

3) Configuración y arranque

src/core/config.py

```
from pydantic_settings import BaseSettings
from pydantic import Field
from typing import Literal
```

```
class Settings(BaseSettings):
    app_name: str = Field(default="Restaurante API")
    env: Literal["dev", "prod", "test"] = "dev"
    db_user: str
    db_password: str
    db_host: str = "127.0.0.1"
    db_port: int = 3306
    db_name: str
    sql_echo: bool = False

    @property
    def sync_dsn(self) -> str:
        return (f"mysql+pymysql://{self.db_user}:{self.db_password}"
                f"@{self.db_host}:{self.db_port}/{self.db_name}?charset=utf8mb4")

    model_config = {"env_file": ".env", "extra": "ignore"}

settings = Settings()
```

---

4) Modelo de datos (ejemplo Plato con SQLAlchemy 2.0)

```
from __future__ import annotations
from decimal import Decimal
from typing import Optional, List
from sqlalchemy import String, Text, Numeric, ForeignKey, Table, Column
from sqlalchemy.orm import Mapped, mapped_column, 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 Plato(Base):
    __tablename__ = "platos"

    id: Mapped[int] = mapped_column(primary_key=True, index=True)
    nombre: Mapped[str] = mapped_column(String(100), index=True)
    precio: Mapped[Decimal] = mapped_column(Numeric(10, 2), nullable=False)
    descripcion: Mapped[Optional[str]] = mapped_column(Text)
    categoria_id: Mapped[int] = mapped_column(ForeignKey("plato_categorias.id"),
index=True)

    alergenos: Mapped[List["Alergeno"]] = relationship(
        secondary=platos_alergenos, back_populates="platos"
    )
```

5) Schemas Pydantic v2 (ejemplo)

```
from decimal import Decimal
from typing import Annotated, Optional, List
from pydantic import BaseModel, Field

Price = Annotated[Decimal, Field(gt=0, max_digits=10, decimal_places=2)]

class PlatoBase(BaseModel):
    nombre: Annotated[str, Field(min_length=1, max_length=100)]
    precio: Price
    descripcion: Optional[str] = None
    categoria_id: int

class PlatoCreate(PlatoBase):
    alergenos_ids: List[int] = []

class PlatoUpdate(BaseModel):
    nombre: Optional[str] = None
    precio: Optional[Price] = None
    descripcion: Optional[str] = None
    categoria_id: Optional[int] = None
    alergenos_ids: Optional[List[int]] = None

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

    model_config = {"from_attributes": True}
```

6) Repositorios y Servicios

Ejemplo de **repositorio** y **servicio** para `Plato`:

```
# repositories/platos.py
from sqlalchemy.orm import Session
from sqlalchemy import select
from src.entities.plato import Plato
from src.entities.alergeno import Alergeno

class PlatoRepository:
    def __init__(self, db: Session): self.db = db

    def create(self, nombre, precio, descripcion, categoria_id, alergenos_ids):
```

```
        plato = Plato(nombre=nombre, precio=precio, descripcion=descripcion,
categoria_id=categoria_id)
        if alergenos_ids:
            alergenos =
self.db.execute(select(Alergeno).where(Alergeno.id.in_(alergenos_ids))).scalars().
all()
            plato.alergenos = alergenos
        self.db.add(plato)
        self.db.commit()
        self.db.refresh(plato)
        return plato
```

```
# services/platos.py
from src.repositories.platos import PlatoRepository

class PlatoService:
    def __init__(self, repo: PlatoRepository): self.repo = repo

    def crear(self, payload):
        return self.repo.create(payload.nombre, payload.precio,
payload.descripcion, payload.categoria_id, payload.alergenos_ids)
```

---

7) Rutas (FastAPI, limpias)

```
from fastapi import APIRouter, Depends, status
from sqlalchemy.orm import Session
from src.api.deps import get_db
from src.schemas.plato import PlatoCreate, PlatoOut
from src.repositories.platos import PlatoRepository
from src.services.platos import PlatoService

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

def get_service(db: Session = Depends(get_db)) -> PlatoService:
    return PlatoService(PlatoRepository(db))

@router.post("", response_model=PlatoOut, status_code=status.HTTP_201_CREATED)
def crear_plato(payload: PlatoCreate, svc: PlatoService = Depends(get_service)):
    return svc.crear(payload)
```

---

8) Otros aspectos clave

- **Errores globales** con manejadores centralizados.
- **Alembic** obligatorio para producción.
- **Seeds idempotentes** en `/scripts/seed.py`.

- **CORS** restringido en producción.
- **Tests** con `pytest` + `httpx.AsyncClient`.
- **Logging INFO en prod**, `echo=True` solo en dev.

---

9) Roadmap recomendado

- **v0**: CRUDs + filtros + seeds.
- **v1**: Repos/Services + Alembic + auth básica.
- **v2**: Búsqueda avanzada + métricas + caching.
