Sí. Divide responsabilidades:

# Stack recomendado (FastAPI)

* **Validación**: Pydantic v2.
* **Auth**: passlib[bcrypt] + JWT corto (access) y refresh con rotación.
* **CSRF** (si usas cookies): fastapi-csrf-protect.
* **CORS**: CORSMiddleware mínimo.
* **Cabeceras/HSTS**: secure.
* **Rate-limit/anti fuerza bruta**: slowapi.
* **Hosts confiables/HTTPS**: TrustedHostMiddleware, HTTPSRedirectMiddleware.
* **Auditoría/errores**: logs estructurados + Sentry.
* **BD**: constraints, índices únicos, usuario con permisos mínimos, TLS.

# Wiring mínimo

from fastapi import FastAPI

from starlette.middleware.httpsredirect import HTTPSRedirectMiddleware

from starlette.middleware.trustedhost import TrustedHostMiddleware

from starlette.middleware.cors import CORSMiddleware

from slowapi import Limiter

from slowapi.util import get\_remote\_address

from slowapi.middleware import SlowAPIMiddleware

from secure import Secure

app = FastAPI()

app.add\_middleware(HTTPSRedirectMiddleware)

app.add\_middleware(TrustedHostMiddleware, allowed\_hosts=["tu.com","www.tu.com"])

app.add\_middleware(CORSMiddleware,

allow\_origins=["https://front.tu.com"],

allow\_credentials=True, allow\_methods=["GET","POST","PUT","DELETE","OPTIONS"],

allow\_headers=["\*"],

)

limiter = Limiter(key\_func=get\_remote\_address); app.state.limiter = limiter

app.add\_middleware(SlowAPIMiddleware)

secure\_headers = Secure(strict\_transport\_security="max-age=31536000; includeSubDomains; preload",

referrer\_policy="strict-origin-when-cross-origin")

@app.middleware("http")

async def set\_headers(req, call\_next):

resp = await call\_next(req); secure\_headers.framework.fastapi(resp); return resp

# Validaciones con Pydantic

from pydantic import BaseModel, Field, field\_validator

from uuid import UUID

class UserCreate(BaseModel):

email: str = Field(pattern=r"^[^@]+@[^@]+\.[^@]+$")

password: str = Field(min\_length=12, max\_length=128)

name: str = Field(min\_length=2, max\_length=80)

@field\_validator("password")

@classmethod

def strong(cls, v):

if v.isnumeric() or v.isalpha(): raise ValueError("password débil")

return v

class UserRead(BaseModel):

id: UUID

email: str

name: str

# Auth segura

from datetime import datetime, timedelta

from passlib.context import CryptContext

import jwt, os

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

JWT\_SECRET = os.environ["JWT\_SECRET"]; ALG = "HS256"

def hash\_pwd(p): return pwd.hash(p)

def verify\_pwd(p, h): return pwd.verify(p, h)

def make\_token(sub: str, minutes: int = 15):

exp = datetime.utcnow() + timedelta(minutes=minutes)

return jwt.encode({"sub": sub, "exp": exp}, JWT\_SECRET, algorithm=ALG)

# Rate-limit en endpoints

from slowapi import Limiter

from slowapi.errors import RateLimitExceeded

from slowapi.util import get\_remote\_address

from fastapi import Depends, HTTPException

limiter = Limiter(key\_func=get\_remote\_address)

@app.get("/login")

@limiter.limit("5/minute")

async def login(): ...

# CSRF (solo si cookies)

from fastapi\_csrf\_protect import CsrfProtect

from pydantic import BaseModel

class CsrfSettings(BaseModel):

secret\_key: str = os.environ["CSRF\_SECRET"]

@CsrfProtect.load\_config

def get\_csrf\_config(): return CsrfSettings()

# Subidas de archivo seguras

from fastapi import UploadFile, HTTPException

ALLOWED = {"image/png","image/jpeg"}; MAX = 2 \* 1024 \* 1024

async def validate\_upload(f: UploadFile):

if f.content\_type not in ALLOWED: raise HTTPException(415, "tipo inválido")

data = await f.read()

if len(data) > MAX: raise HTTPException(413, "archivo grande")

await f.seek(0)

# BD segura

* ORM con parámetros. Nunca concatenar SQL.
* UNIQUE y CHECK (ej. porcentajes 0–100).
* Usuario DB sin SUPERUSER. TLS (sslmode=require).
* Migraciones controladas.

# Checklist operativo

* HTTPS + HSTS activo.
* CORS mínimo. CSRF si cookies.
* Cookies HttpOnly + Secure + SameSite=Lax/Strict.
* JWT cortos. Refresh con rotación y revocación.
* Rate-limit en login y rutas sensibles.
* Logs/auditoría de auth y cambios críticos.
* Backups cifrados y pruebas de restore.
* Dependencias actualizadas y escaneadas.

¿Quieres que te deje un repo base con este wiring y ejemplos listos?

##### Dijiste:

Y donde está el XSS y el SQL INJECTION?

###### ChatGPT dijo:

Pensado durante 29s

Cubre ambos así:

# XSS (inyección de JavaScript)

**Principios**

* Nunca renderices HTML no confiable sin sanitizar.
* Activa autoescape en plantillas. Evita dangerouslySetInnerHTML en React.
* Añade CSP fuerte y cabeceras seguras.
* Sirve descargas como attachment, no inline.

**Código**

# Middleware de cabeceras seguras + CSP

from secure import Secure

secure\_headers = Secure(

content\_security\_policy="default-src 'self'; script-src 'self'; object-src 'none'; base-uri 'self'; frame-ancestors 'none'",

referrer\_policy="strict-origin-when-cross-origin",

content\_type\_nosniff=True,

strict\_transport\_security="max-age=31536000; includeSubDomains; preload",

)

@app.middleware("http")

async def set\_headers(req, call\_next):

r = await call\_next(req)

secure\_headers.framework.fastapi(r)

return r

# Sanitizar HTML de usuario (si permites rich text)

import bleach

SAFE\_TAGS = ["b","i","strong","em","a","p","ul","ol","li","br","span"]

SAFE\_ATTRS = {"a":["href","rel","target"], "span":["class"]}

def sanitize\_html(html: str) -> str:

return bleach.clean(html, tags=SAFE\_TAGS, attributes=SAFE\_ATTRS, protocols=["http","https","mailto"])

# En tu schema Pydantic v2

from pydantic import BaseModel, field\_validator

class CommentCreate(BaseModel):

body\_html: str

@field\_validator("body\_html")

@classmethod

def \_san(cls, v): return sanitize\_html(v)

# Plantillas Jinja2 con autoescape

from fastapi.templating import Jinja2Templates

templates = Jinja2Templates("templates")

templates.env.autoescape = True # .html ya viene autoescapado

# SQL Injection

**Principios**

* Usa SQLAlchemy ORM/SQL con **parámetros**, nunca concatenes strings.
* Lista blanca para order\_by, fields, y filtros dinámicos.
* Escapa comodines en LIKE. Limita permisos del usuario DB.

**Código**

# Seguro con ORM

from sqlalchemy import select

stmt = select(User).where(User.email == email)

rows = (await db.execute(stmt)).scalars().all()

# Seguro con SQL crudo parametrizado

from sqlalchemy import text, bindparam

stmt = text("SELECT \* FROM users WHERE email = :email").bindparams(bindparam("email", email))

rows = (await db.execute(stmt)).mappings().all()

# Lista blanca para ordenamiento

ALLOWED\_SORT = {"name": User.name, "created\_at": User.created\_at}

col = ALLOWED\_SORT.get(sort or "created\_at") # ignora cualquier otro valor

stmt = select(User).order\_by(col.desc() if desc else col.asc())

# Búsqueda LIKE segura (escapa % y \_)

term = term.replace("\\","\\\\").replace("%","\\%").replace("\_","\\\_")

stmt = select(Product).where(Product.name.ilike(f"%{term}%", escape="\\"))

# Usuario DB con mínimos permisos + TLS

# Ejemplo Postgres URI: postgresql+asyncpg://appuser:\*\*\*@host/db?sslmode=require

# Checklist rápido

* CSP estricta y X-Content-Type-Options: nosniff.
* Autoescape en plantillas. Sanitización si aceptas HTML.
* No innerHTML ni dangerouslySetInnerHTML salvo contenido sanitizado.
* ORM/SQL siempre parametrizado. Whitelists para sort/fields.
* Escapa comodines en LIKE. Sin concatenaciones.
* Usuario DB sin SUPERUSER. TLS a la BD.
* Logs y rate-limit en login. Cookies HttpOnly + Secure si usas cookies.