Objetivo: Diseñar e implementar una capa de ciberseguridad robusta, centralizada y profesional para el backend FastAPI, asegurando que todas las funcionalidades sigan las mejores prácticas de la industria (OWASP, NIST). Contexto: Se requiere integrar medidas de seguridad de manera cohesiva y no redundante, priorizando la mantenibilidad y la escalabilidad. La seguridad debe ser una capa transversal, no una implementación ad-hoc en cada endpoint. Requisitos Técnicos Específicos:

1. Autenticación & Autorización Centralizada: Implementar un middleware de autenticación JWT (OAuth2 con Password Flow) que valide tokens en cada request. Crear un sistema de roles y permisos (RBAC) con políticas definidas en un solo lugar (ej: security/authorization.py). Usar decoradores personalizados (ej: @requires\_role('admin')) para proteger endpoints de manera declarativa.

2. Validación de Datos de Entrada (Input): Extender el uso de Pydantic no solo para serialización, sino para validación exhaustiva de todos los request bodies, query parameters y path parameters. Implementar sanitización centralizada contra inyecciones (SQL, XSS) para todos los campos de texto. Validar tipos de archivo y tamaño máximo en endpoints de upload.

3. Protección de Endpoints Común: Implementar rate limiting global (ej: usando slowapi o un middleware) para mitigar ataques de fuerza bruta y DDoS. Configurar CORS de manera restrictiva, especificando orígenes permitidos de forma explícita. Integración con Helmet (o sus equivalentes en FastAPI) para headers de seguridad HTTP (HSTS, X-Content-Type-Options, etc.).

4. Seguridad de la Base de Datos: Usar únicamente consultas parametrizadas o ORM (SQLAlchemy) para prevenir toda forma de inyección SQL. Centralizar el manejo de conexiones a la BD y el logging de queries sensibles.

5. Manejo Centralizado de Errores y Logging: Crear un manejador global de excepciones que no filtre información sensible (stack traces, detalles de BD) en respuestas de producción. Implementar un sistema de logging estructurado que registre eventos de seguridad (logins fallidos, accesos denegados, errores de validación) para su posterior auditoría. Clasificar los logs por niveles (INFO, WARNING, ERROR) y canalizarlos a un destino seguro (ej: archivo, Syslog, SIEM).

6. Gestión de Secretos: No hardcodear credenciales, API keys o secrets en el código. Leer todas las variables sensibles desde variables de entorno o un vault de secrets, usando una biblioteca como pydantic-settings.

7. Otras Hardening: Implementar hashing robusto (bcrypt) con salt para todas las contraseñas. Validar y sanear todas las URLs usadas en redirecciones para evitar Open Redirects. Revisar y asegurar las dependencias del proyecto (requirements.txt) para evitar librerías con vulnerabilidades conocidas. Entregables Esperados: Análisis: Un resumen de los riesgos de seguridad mitigados y la arquitectura propuesta. Código: La implementación de los módulos centralizados en una estructura de carpetas como: text /app /security \_\_init\_\_.py authentication.py # Middleware JWT authorization.py # Lógica de RBAC input\_validation.py # Sanitización rate\_limiting.py # Config de rate limits /core config.py # Configuración con pydantic-settings logging.py # Configuración centralizada de logging /middleware security\_headers.py # Middleware para headers de seguridad /utils security\_utils.py # Funciones para hashing, etc. Documentación: Comentarios en el código y/o un README que explique cómo se integra cada capa de seguridad y cómo añadir nuevas políticas. Materiales Proporcionados: Se adjunta el código actual del backend en formato ZIP para su análisis y para que la implementación sea coherente con la base de código existente. Por favor suministrarme el código para copiar y pegar y explicar cada recurso como si yo estuviera apenas aprendiendo.

**ChatGPT dijo:**

Pensado durante 18s

Resumen de arquitectura y riesgos mitigados:

* AutN/Z central: JWT en middleware + RBAC declarativo. Mitiga: sesión inválida, escalamiento de privilegios.
* Validación entrada: Pydantic v2 + sanitización central. Mitiga: XSS, SQLi, path traversal básico.
* Endpoints protegidos: rate limit + CORS restrictivo + headers seguros. Mitiga: fuerza bruta, CSRF básico, clickjacking, sniffing.
* BD segura: ORM + logging de consultas. Mitiga: inyección y fuga de datos.
* Errores y logging: handlers globales + logging estructurado. Mitiga: filtrado de secretos y trazas en prod, auditoría.
* Secretos: pydantic-settings. Mitiga: secretos en repo.
* Hardening: bcrypt, comparación segura, validación de URLs. Mitiga: cracking y open redirects.

A continuación, pega los archivos propuestos. Están listos para copiar y usar.

**/app/core/config.py**

from pydantic\_settings import BaseSettings, SettingsConfigDict

from typing import List, Optional

from pathlib import Path

class Settings(BaseSettings):

# App

APP\_NAME: str = "POS API"

ENV: str = "dev" # dev|stg|prod

DEBUG: bool = True

# Security

JWT\_SECRET: str

JWT\_ALG: str = "HS256"

ACCESS\_TOKEN\_EXPIRE\_MINUTES: int = 60

EXEMPT\_PATHS: List[str] = ["/health", "/docs", "/openapi.json", "/auth/token"]

# CORS

CORS\_ALLOW\_ORIGINS: List[str] = ["http://localhost:3000"]

CORS\_ALLOW\_METHODS: List[str] = ["GET", "POST", "PUT", "PATCH", "DELETE", "OPTIONS"]

CORS\_ALLOW\_HEADERS: List[str] = ["Authorization", "Content-Type"]

CORS\_ALLOW\_CREDENTIALS: bool = True

# Rate limit

RATE\_LIMIT\_ENABLED: bool = True

RATE\_LIMIT\_REQUESTS: int = 100

RATE\_LIMIT\_WINDOW\_SECONDS: int = 60

# Headers

HSTS\_MAX\_AGE: int = 31536000 # 1 año

CONTENT\_SECURITY\_POLICY: str = "default-src 'self'; img-src 'self' data:; script-src 'self'; style-src 'self' 'unsafe-inline'"

# Logging

LOG\_LEVEL: str = "INFO"

LOG\_JSON: bool = False

LOG\_SQLALCHEMY: bool = False

# DB

DATABASE\_URL: Optional[str] = None # ya la tienes en tu proyecto

model\_config = SettingsConfigDict(env\_file=(".env", ".env.local"), case\_sensitive=False)

settings = Settings() # importa desde app.core.config

**/app/core/logging.py**

import logging, sys, json

from typing import Any, Dict

from fastapi import Request

from fastapi.responses import JSONResponse

from starlette import status

from app.core.config import settings

def setup\_logging() -> None:

root = logging.getLogger()

root.handlers.clear()

level = getattr(logging, settings.LOG\_LEVEL.upper(), logging.INFO)

root.setLevel(level)

class JsonFormatter(logging.Formatter):

def format(self, record: logging.LogRecord) -> str:

base = {

"level": record.levelname,

"logger": record.name,

"msg": record.getMessage(),

}

if record.exc\_info:

base["exc\_info"] = self.formatException(record.exc\_info)

return json.dumps(base, ensure\_ascii=False)

h = logging.StreamHandler(sys.stdout)

h.setLevel(level)

h.setFormatter(JsonFormatter() if settings.LOG\_JSON else logging.Formatter("%(levelname)s %(name)s: %(message)s"))

root.addHandler(h)

if settings.LOG\_SQLALCHEMY:

logging.getLogger("sqlalchemy.engine").setLevel(logging.INFO)

# ---- Exception Handlers (regístralos en main) ----

async def http\_exception\_handler(request: Request, exc):

# FastAPI ya hace esto, pero aquí filtramos detalle en prod si fuera necesario

return JSONResponse(status\_code=exc.status\_code, content={"detail": exc.detail})

async def unhandled\_exception\_handler(request: Request, exc: Exception):

logging.getLogger(\_\_name\_\_).exception("Unhandled error")

code = status.HTTP\_500\_INTERNAL\_SERVER\_ERROR

msg = "Internal server error" if settings.ENV == "prod" else str(exc)

return JSONResponse(status\_code=code, content={"detail": msg})

**/app/middleware/security\_headers.py**

from starlette.middleware.base import BaseHTTPMiddleware

from starlette.requests import Request

from starlette.responses import Response

from app.core.config import settings

class SecurityHeadersMiddleware(BaseHTTPMiddleware):

async def dispatch(self, request: Request, call\_next):

resp: Response = await call\_next(request)

resp.headers.setdefault("X-Content-Type-Options", "nosniff")

resp.headers.setdefault("X-Frame-Options", "DENY")

resp.headers.setdefault("Referrer-Policy", "no-referrer")

resp.headers.setdefault("X-XSS-Protection", "0")

resp.headers.setdefault("Strict-Transport-Security", f"max-age={settings.HSTS\_MAX\_AGE}; includeSubDomains")

resp.headers.setdefault("Permissions-Policy", "geolocation=()")

resp.headers.setdefault("Content-Security-Policy", settings.CONTENT\_SECURITY\_POLICY)

return resp

**/app/utils/security\_utils.py**

import re, hmac, secrets

from typing import Any, Dict

from bcrypt import gensalt, hashpw, checkpw

# Sanitización básica central

BLOCK\_PATTERNS = [

re.compile(r"<\s\*script", re.I),

re.compile(r"on\w+\s\*=", re.I),

re.compile(r"(?:;|\b)drop\s+table\b", re.I),

re.compile(r"(?:;|\b)union\s+select\b", re.I),

]

FIELD\_MAX\_LENGTHS = {"code": 10, "name": 100, "description": 500}

class SecurityUtils:

@staticmethod

def sanitize\_input(value: str, field: str | None = None) -> str:

v = value.strip()

for pat in BLOCK\_PATTERNS:

v = pat.sub("", v)

if field and field in FIELD\_MAX\_LENGTHS:

v = v[:FIELD\_MAX\_LENGTHS[field]]

return v

@staticmethod

def sanitize\_data(data: Dict[str, Any]) -> Dict[str, Any]:

out: Dict[str, Any] = {}

for k, v in data.items():

if isinstance(v, str):

out[k] = SecurityUtils.sanitize\_input(v, k)

elif isinstance(v, dict):

out[k] = SecurityUtils.sanitize\_data(v)

elif isinstance(v, list):

out[k] = [SecurityUtils.sanitize\_data(i) if isinstance(i, dict) else (SecurityUtils.sanitize\_input(i, k) if isinstance(i, str) else i) for i in v]

else:

out[k] = v

return out

# Password hashing

@staticmethod

def hash\_password(password: str) -> str:

return hashpw(password.encode("utf-8"), gensalt()).decode("utf-8")

@staticmethod

def verify\_password(password: str, hashed: str) -> bool:

try:

return checkpw(password.encode("utf-8"), hashed.encode("utf-8"))

except Exception:

return False

@staticmethod

def secure\_compare(a: str, b: str) -> bool:

return hmac.compare\_digest(a, b)

@staticmethod

def is\_safe\_url(url: str, allowed\_hosts: list[str]) -> bool:

# evita open redirect: solo hosts permitidos

try:

from urllib.parse import urlparse

netloc = urlparse(url).netloc

return netloc == "" or netloc in allowed\_hosts

except Exception:

return False

**/app/security/authentication.py (JWT global + helpers)**

import time, logging

from typing import Optional, Dict

from starlette.middleware.base import BaseHTTPMiddleware

from starlette.requests import Request

from starlette.responses import Response, JSONResponse

from fastapi import Depends, HTTPException, status

from fastapi.security import OAuth2PasswordBearer

from jose import jwt, JWTError

from app.core.config import settings

logger = logging.getLogger(\_\_name\_\_)

oauth2\_scheme = OAuth2PasswordBearer(tokenUrl="/auth/token") # tu endpoint de login

def create\_access\_token(sub: str, claims: Optional[Dict] = None, expires\_minutes: int | None = None) -> str:

to\_encode = {"sub": sub, "iat": int(time.time())}

if claims:

to\_encode.update(claims)

exp = int(time.time()) + 60 \* (expires\_minutes or settings.ACCESS\_TOKEN\_EXPIRE\_MINUTES)

to\_encode.update({"exp": exp})

return jwt.encode(to\_encode, settings.JWT\_SECRET, algorithm=settings.JWT\_ALG)

def decode\_token(token: str) -> dict:

return jwt.decode(token, settings.JWT\_SECRET, algorithms=[settings.JWT\_ALG])

class JWTAuthMiddleware(BaseHTTPMiddleware):

def \_\_init\_\_(self, app, exempt\_paths: Optional[set[str]] = None):

super().\_\_init\_\_(app)

self.exempt = set(settings.EXEMPT\_PATHS) | set(exempt\_paths or [])

async def dispatch(self, request: Request, call\_next):

path = request.url.path

if any(path.startswith(p) for p in self.exempt):

return await call\_next(request)

auth = request.headers.get("Authorization", "")

if not auth.startswith("Bearer "):

return JSONResponse(status\_code=status.HTTP\_401\_UNAUTHORIZED, content={"detail": "Not authenticated"})

token = auth.split(" ", 1)[1]

try:

payload = decode\_token(token)

request.state.user = {

"sub": payload.get("sub"),

"roles": payload.get("roles", []),

"perms": payload.get("perms", {}),

}

return await call\_next(request)

except JWTError:

return JSONResponse(status\_code=status.HTTP\_401\_UNAUTHORIZED, content={"detail": "Invalid token"})

# Dependencia por si quieres usarla en endpoints

async def get\_current\_user(request: Request, token: str = Depends(oauth2\_scheme)) -> dict:

# Si el middleware ya puso el usuario, úsalo

if getattr(request.state, "user", None):

return request.state.user

# Fallback: valida token

payload = decode\_token(token)

return {

"sub": payload.get("sub"),

"roles": payload.get("roles", []),

"perms": payload.get("perms", {}),

}

**/app/security/authorization.py (RBAC central)**

from typing import Callable, Iterable, Optional

from fastapi import Depends, HTTPException, status

from starlette.requests import Request

def \_has\_any(owned: Iterable[str], required: Iterable[str]) -> bool:

s = set(owned or [])

return any(r in s for r in required)

def \_ensure\_denied():

raise HTTPException(status\_code=status.HTTP\_403\_FORBIDDEN, detail="Forbidden")

def require\_roles(\*roles: str):

async def \_dep(request: Request):

user = getattr(request.state, "user", None)

user\_roles = (user or {}).get("roles", [])

if not \_has\_any(user\_roles, roles):

\_ensure\_denied()

return True

return \_dep

def require\_permissions(\*perms: str):

async def \_dep(request: Request):

user = getattr(request.state, "user", None)

user\_perms = (user or {}).get("perms", {})

have = set(k for k, v in (user\_perms or {}).items() if v is True)

if not \_has\_any(have, perms):

\_ensure\_denied()

return True

return \_dep

# Decoradores opcionales (a costa de legibilidad). Recomendado usar Depends(require\_roles("admin"))

def requires\_role(\*roles: str):

def wrapper(fn: Callable):

async def inner(\*args, \*\*kwargs):

request: Request = kwargs.get("request") or next((a for a in args if isinstance(a, Request)), None)

if request is None:

\_ensure\_denied()

await require\_roles(\*roles)(request) # lanzará 403 si falla

return await fn(\*args, \*\*kwargs)

return inner

return wrapper

**/app/security/input\_validation.py (middleware y utilidades)**

import logging, re

from starlette.middleware.base import BaseHTTPMiddleware

from starlette.requests import Request

from starlette.responses import Response, JSONResponse

from app.utils.security\_utils import SecurityUtils

logger = logging.getLogger(\_\_name\_\_)

SUSPICIOUS = re.compile(r"(<\s\*script|union\s+select|drop\s+table|on\w+\s\*=)", re.I)

class InputSanitizerMiddleware(BaseHTTPMiddleware):

async def dispatch(self, request: Request, call\_next):

qs = request.url.query

if qs and SUSPICIOUS.search(qs):

logger.warning("Suspicious querystring blocked")

return JSONResponse(status\_code=400, content={"detail": "Bad querystring"})

# No mutamos el body aquí para no romper firmas. Los modelos Pydantic sanitizan.

return await call\_next(request)

**/app/security/rate\_limiting.py (sin dependencias externas; in-memory)**

import time

from typing import Dict, Tuple

from starlette.middleware.base import BaseHTTPMiddleware

from starlette.requests import Request

from starlette.responses import JSONResponse

from app.core.config import settings

Bucket = Tuple[int, int] # (recuento, ventana\_inicio)

class RateLimitMiddleware(BaseHTTPMiddleware):

def \_\_init\_\_(self, app):

super().\_\_init\_\_(app)

self.store: Dict[str, Bucket] = {}

async def dispatch(self, request: Request, call\_next):

if not settings.RATE\_LIMIT\_ENABLED:

return await call\_next(request)

key = f"{request.client.host}:{request.url.path}"

now = int(time.time())

count, window\_start = self.store.get(key, (0, now))

if now - window\_start >= settings.RATE\_LIMIT\_WINDOW\_SECONDS:

count, window\_start = 0, now

count += 1

self.store[key] = (count, window\_start)

if count > settings.RATE\_LIMIT\_REQUESTS:

retry = settings.RATE\_LIMIT\_WINDOW\_SECONDS - (now - window\_start)

return JSONResponse(status\_code=429, content={"detail": "Too Many Requests", "retry\_after": retry})

return await call\_next(request)

**Integración en app/main.py (añade o ajusta)**

# app/main.py (fragmento clave)

from fastapi import FastAPI

from fastapi.middleware.cors import CORSMiddleware

from app.core.config import settings

from app.core.logging import setup\_logging, http\_exception\_handler, unhandled\_exception\_handler

from app.security.authentication import JWTAuthMiddleware

from app.security.input\_validation import InputSanitizerMiddleware

from app.security.rate\_limiting import RateLimitMiddleware

from app.middleware.security\_headers import SecurityHeadersMiddleware

def create\_app() -> FastAPI:

setup\_logging()

app = FastAPI(title=settings.APP\_NAME, debug=settings.DEBUG)

# CORS restrictivo

app.add\_middleware(

CORSMiddleware,

allow\_origins=settings.CORS\_ALLOW\_ORIGINS,

allow\_credentials=settings.CORS\_ALLOW\_CREDENTIALS,

allow\_methods=settings.CORS\_ALLOW\_METHODS,

allow\_headers=settings.CORS\_ALLOW\_HEADERS,

)

# Middlewares de seguridad (orden importa)

app.add\_middleware(SecurityHeadersMiddleware)

app.add\_middleware(InputSanitizerMiddleware)

app.add\_middleware(RateLimitMiddleware)

app.add\_middleware(JWTAuthMiddleware) # valida JWT en cada request

# Handlers globales

from fastapi import HTTPException

app.add\_exception\_handler(HTTPException, http\_exception\_handler)

app.add\_exception\_handler(Exception, unhandled\_exception\_handler)

# monta routers existentes…

# from app.routers import brand, category, ...

# app.include\_router(brand.router)

# app.include\_router(category.router)

# ...

return app

app = create\_app()

**Uso básico de RBAC en endpoints**

from fastapi import APIRouter, Depends

from app.security.authorization import require\_roles, require\_permissions

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

@router.get("/stats", dependencies=[Depends(require\_roles("admin"))])

async def stats():

return {"ok": True}

@router.delete("/danger", dependencies=[Depends(require\_permissions("can\_delete"))])

async def danger():

return {"done": True}

**Notas operativas**

1. Generación de tokens:

from app.security.authentication import create\_access\_token

jwt = create\_access\_token("user-uuid", claims={"roles": ["admin"], "perms": {"can\_delete": True}})

1. Contraseñas:

from app.utils.security\_utils import hash\_password, verify\_password

hashed = hash\_password("S3guro\*123")

verify\_password("S3guro\*123", hashed) # True

1. Redirecciones seguras:

from app.utils.security\_utils import is\_safe\_url

is\_safe\_url(next\_url, allowed\_hosts=["tudominio.com"])

**Checklist rápida**

* [.env] define JWT\_SECRET, CORS\_ALLOW\_ORIGINS.
* pip install python-jose[cryptography] bcrypt pydantic-settings.
* Incluye middlewares y handlers en main.py.
* Asegura que tus Pydantic Schemas heredan SecureBaseModel y sanitizan.
* Activa LOG\_SQLALCHEMY si quieres ver queries en dev.

Con esto tienes una capa de ciberseguridad central, mantenible y homogénea.

**Dijiste:**

Por favor explicar con más detalle la función o que resuelve cada uno de estos recursos, como si estuviera iniciando.

**ChatGPT dijo:**

Pensado durante unos segundos

Aquí tienes qué hace cada pieza, qué problema resuelve y cómo usarla. Directo y sin rodeos.

**/app/core/config.py — configuración central**

* **Qué es**: un objeto Settings que lee variables desde .env usando pydantic-settings.
* **Problema que resuelve**: evita “hardcodear” secretos y parámetros. Un solo lugar para toggles de seguridad.
* **Claves**: JWT\_SECRET, CORS, rate limit, HSTS, CSP, nivel de logs.
* **Cómo usar**: importa from app.core.config import settings. Cambia valores en .env.
* **Errores típicos**: falta JWT\_SECRET → 401. CORS mal configurado → el frontend no puede llamar.

**/app/core/logging.py — logging y manejo global de errores**

* **Qué es**: inicializa formato de logs (texto o JSON) y define handlers para excepciones.
* **Problema que resuelve**: logs consistentes y sin filtrar datos sensibles en producción.
* **Claves**: setup\_logging() se llama en main.py. http\_exception\_handler y unhandled\_exception\_handler devuelven respuestas limpias.
* **Cómo usar**: llama setup\_logging() al crear la app. Ajusta LOG\_JSON, LOG\_LEVEL en .env.
* **Errores típicos**: loggear exc\_info en prod con detalle sensible. Aquí se filtra según ENV.

**/app/middleware/security\_headers.py — headers de seguridad**

* **Qué es**: middleware que añade HSTS, X-Frame-Options, nosniff, CSP, etc.
* **Problema que resuelve**: mitiga clickjacking, sniffing, inyección de contenido.
* **Claves**: Content-Security-Policy restrictiva por defecto.
* **Cómo usar**: app.add\_middleware(SecurityHeadersMiddleware) en main.py.
* **Errores típicos**: CSP muy restrictiva rompe assets del frontend. Ajusta CONTENT\_SECURITY\_POLICY en .env.

**/app/utils/security\_utils.py — utilidades de seguridad**

* **Qué es**: sanitizado de strings, hashing de contraseñas bcrypt, comparación constante, validación de URLs.
* **Problema que resuelve**: XSS básico, SQLi por concatenación, open redirects, gestión segura de passwords.
* **Claves**: sanitize\_input, sanitize\_data, hash\_password, verify\_password, is\_safe\_url.
* **Cómo usar**: se invoca desde modelos Pydantic y lógicas de negocio.
* **Errores típicos**: olvidar verify\_password y comparar strings directamente.

**/app/security/authentication.py — autenticación JWT central**

* **Qué es**: middleware que valida Authorization: Bearer … en cada request. Helpers para crear/validar tokens.
* **Problema que resuelve**: acceso no autorizado y tokens inválidos.
* **Claves**: JWTAuthMiddleware, create\_access\_token, decode\_token, get\_current\_user.
* **Cómo usar**: añade el middleware en main.py. Genera tokens al autenticar usuarios y coloca roles y perms en el payload.
* **Errores típicos**: token con alg distinto a settings.JWT\_ALG. JWT\_SECRET incorrecto → 401.

**/app/security/authorization.py — RBAC (roles y permisos)**

* **Qué es**: dependencias para exigir roles o permisos en endpoints.
* **Problema que resuelve**: control de acceso declarativo y consistente.
* **Claves**: require\_roles("admin"), require\_permissions("can\_delete"). Devuelven 403 si no cumple.
* **Cómo usar**: dependencies=[Depends(require\_roles("admin"))] en rutas. Agrega roles/perms al JWT.
* **Errores típicos**: olvidar poblar roles/perms al emitir el token. Resultado: 403.

**/app/security/input\_validation.py — validación/sanitización transversal**

* **Qué es**: middleware que detecta querystrings sospechosos y bloquea. El resto se sanitiza en Pydantic.
* **Problema que resuelve**: reduce vectores básicos de XSS/SQLi en parámetros.
* **Claves**: regex de patrones peligrosos; respuesta 400 si detecta payload sospechoso.
* **Cómo usar**: app.add\_middleware(InputSanitizerMiddleware).
* **Errores típicos**: falsos positivos si pasas cadenas que matchean el regex. Ajusta la expresión si es necesario.

**/app/security/rate\_limiting.py — rate limit global**

* **Qué es**: middleware in-memory de “balde” por IP+ruta.
* **Problema que resuelve**: fuerza bruta y abuso de endpoints.
* **Claves**: ventana y umbral configurables en .env. Devuelve 429 con retry\_after.
* **Cómo usar**: app.add\_middleware(RateLimitMiddleware).
* **Errores típicos**: in-memory no sirve para múltiples instancias. Para prod usa Redis u otra store.

**/app/routers/catalog\_router.py — factory de routers**

* **Qué es**: genera routers CRUD con el mismo patrón de validación, respuestas y errores.
* **Problema que resuelve**: evita lógica duplicada en cada endpoint. Estándar único.
* **Claves**: recibe tipos Pydantic (SCreate, SRead…) y un CRUD concreto. Soporta /import opcional.
* **Cómo usar**: crea router = build\_catalog\_router(...) en cada archivo de router.
* **Errores típicos**: tipos forward-ref si activas from \_\_future\_\_ import annotations. No lo uses aquí.

**/app/crud/catalog\_crud.py — CRUD genérico seguro**

* **Qué es**: clase CatalogCRUD con create/list/get/update/patch/delete/import\_csv\_text.
* **Problema que resuelve**: validaciones y manejo de errores homogéneos como en PaymentTerm.
* **Claves**: cheques de unicidad, sanitizado central, auditoría opcional, commit/rollback controlados en router.
* **Cómo usar**: crea \_crud = CatalogCRUD(Model, ...) por entidad y reexporta funciones.
* **Errores típicos**: olvidar await db.commit() en routers. El factory ya lo hace.

**Schemas con SecureBaseModel y EntityBase**

* **Qué es**: bases Pydantic que sanitizan todo y fijan límites de longitud y regex de code.
* **Problema que resuelve**: entradas coherentes y libres de payloads peligrosos.
* **Claves**: EntityBase con code/name/description/active. Patch/Read/ListResponse heredan SecureBaseModel.
* **Cómo usar**: cada \*Base hereda EntityBase. Los \*Patch y lecturas heredan SecureBaseModel.
* **Errores típicos**: falta de alias en schemas si tu router usa nombres antiguos (ej. SubGroup\*). Añade alias.

**Integración en main.py — orden recomendado**

1. setup\_logging().
2. CORS.
3. SecurityHeadersMiddleware.
4. InputSanitizerMiddleware.
5. RateLimitMiddleware.
6. JWTAuthMiddleware.
7. Handlers globales.
8. include\_router(...).

Orden importa. Primero cabezales, luego sanitizado, luego limitación, luego autenticación.

**Flujo típico de una petición**

1. Llega el request.
2. Headers de seguridad se aplican.
3. Se valida querystring básico.
4. Se evalúa rate limit.
5. Se valida JWT y se carga request.state.user.
6. Router ejecuta. Pydantic valida y sanitiza body.
7. CRUD hace cheques de unicidad y DB segura.
8. Respuesta con modelo de salida consistente.
9. Logs estructurados. Errores limpios si ocurren.

**Cómo añadir una política nueva (ejemplo)**

* Nuevo permiso: can\_export.
* Al emitir token: claims={"perms": {"can\_export": True}}.
* En endpoint:
* from fastapi import Depends
* from app.security.authorization import require\_permissions
* @router.get("/export", dependencies=[Depends(require\_permissions("can\_export"))])
* async def export():
* ...

**Cómo exigir rol admin**

from fastapi import Depends

from app.security.authorization import require\_roles

@router.delete("/{id}", dependencies=[Depends(require\_roles("admin"))])

async def remove(id: UUID):

...

**Validación de archivos en upload**

* Usa UploadFile. Valida tipo y tamaño antes de leer:
* @router.post("/import")
* async def import\_csv(file: UploadFile = File(...)):
* if file.content\_type not in ("text/csv", "application/vnd.ms-excel"):
* raise HTTPException(400, "Tipo no permitido")
* data = await file.read()
* if len(data) > 2\_000\_000: # 2MB
* raise HTTPException(413, "Archivo demasiado grande")
* ...

**Mapeo rápido a OWASP/NIST**

* A01: Broken Access Control → RBAC (authorization.py) + dependencias.
* A07: Identification/Authentication Failures → JWT middleware (authentication.py), bcrypt.
* A03: Injection → ORM + sanitizado (security\_utils.py, schemas).
* A05: Security Misconfiguration → headers, CORS, variables en .env.
* A08: Software/Data Integrity → pin de dependencias y revisión de requirements.txt.
* A09: Logging/Monitoring → core/logging.py con eventos de seguridad.

**Checklist de despliegue**

* .env con JWT\_SECRET fuerte y ENV=prod.
* CORS\_ALLOW\_ORIGINS con dominios reales.
* CSP ajustado a tu frontend.
* Rate limit activado y calibrado.
* Logs en JSON si vas a SIEM.
* Revisar dependencias con pip-audit o safety.

Con esto entiendes el “para qué” de cada recurso y cómo aplicarlo sin sorpresas. Si quieres, te doy ejemplos concretos pegados a una entidad específica de tu repo.

Principio del formulario



Final del formulario

ChatGPT puede cometer errores. Comprueba la información importante.