MÓDULO 11: FASTAPI PROFESIONAL

Async, Dependency Injection, Middleware y Error Handling

Guía MLOps v5.0: Senior Edition | DuqueOM | Noviembre 2025

⚡ MÓDULO 11: FastAPI Profesional

APIs que Escalan y se Mantienen

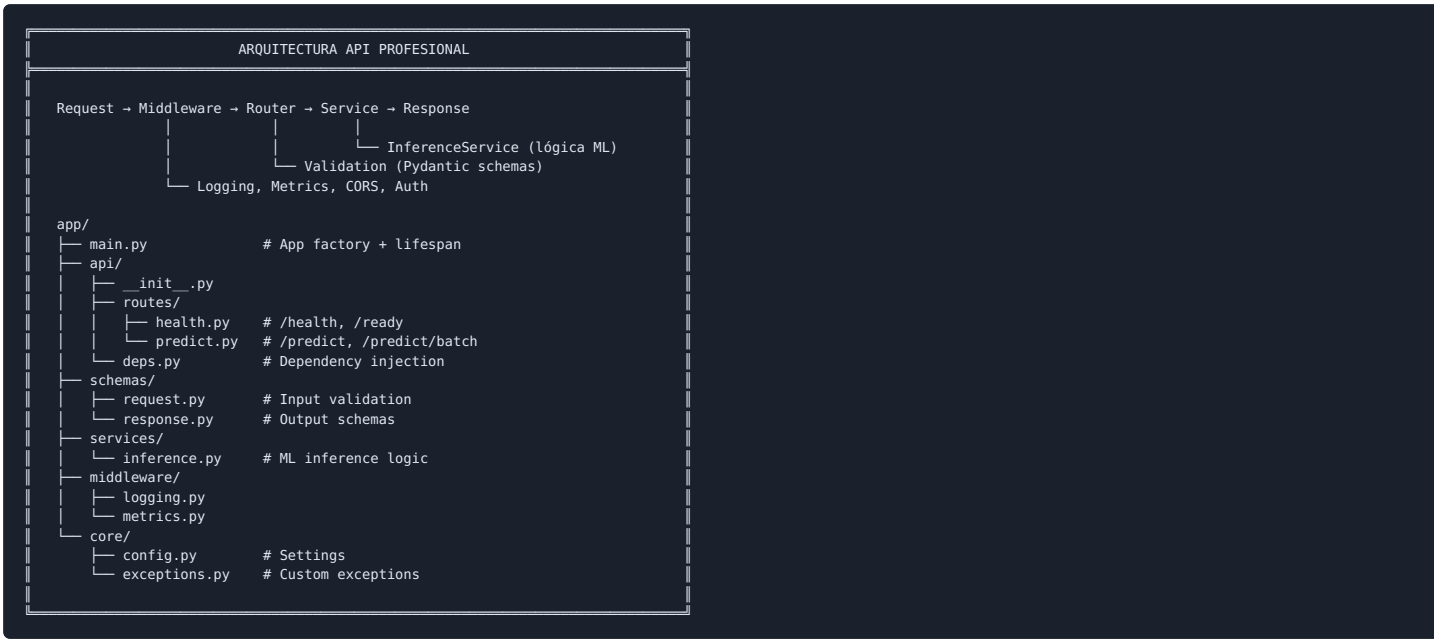
"Una API sin validación es una invitación al caos."

Duración	Teoría	Práctica
5-6 horas	25%	75%

Lo Que Lograrás en Este Módulo

1. **Diseñar** APIs con arquitectura limpia
2. **Implementar** validación robusta con Pydantic
3. **Configurar** middleware para logging y métricas
4. **Manejar** errores de forma consistente

11.1 Arquitectura de la API



11.2 Schemas con Pydantic

```python
# app/schemas/request.py
from pydantic import BaseModel, Field, field_validator
from typing import Literal, List
from enum import Enum

class Geography(str, Enum):
    FRANCE = "France"
    GERMANY = "Germany"
    SPAIN = "Spain"

class Gender(str, Enum):
    MALE = "Male"
    FEMALE = "Female"

class PredictionRequest(BaseModel):
    """Request para predicción individual."""

    credit_score: int = Field(
        ...,
        ge=300,
        le=850,
        description="Score crediticio del cliente",
        examples=[650]
    )
    age: int = Field(..., ge=18, le=100, examples=[35])
    tenure: int = Field(..., ge=0, le=50, description="Años como cliente")
    balance: float = Field(..., ge=0, examples=[50000.0])
    num_of_products: int = Field(..., ge=1, le=4, alias="numOfProducts")
    has_cr_card: bool = Field(..., alias="hasCrCard")
    is_active_member: bool = Field(..., alias="isActiveMember")
    estimated_salary: float = Field(..., ge=0, alias="estimatedSalary")
    geography: Geography
    gender: Gender

    @field_validator('balance')
    @classmethod
    def validate_balance(cls, v: float) -> float:
        if v < 0:
            raise ValueError('Balance no puede ser negativo')
        return round(v, 2)

    class Config:
        populate_by_name = True  # Acepta tanto snake_case como camelCase
        json_schema_extra = {
            "example": {
                "credit_score": 650,
                "age": 35,
                "tenure": 5,
                "balance": 50000.0,
                "num_of_products": 2,
                "has_cr_card": True,
                "is_active_member": True,
                "estimated_salary": 75000.0,
                "geography": "France",
                "gender": "Female"
            }
        }


class BatchPredictionRequest(BaseModel):
    """Request para predicción en batch."""
    instances: List[PredictionRequest] = Field(
        ...,
        min_length=1,
        max_length=1000,
        description="Lista de instancias a predecir"
    )
```

```python
# app/schemas/response.py
from pydantic import BaseModel, Field
from typing import List, Literal, Optional
from datetime import datetime

class PredictionResponse(BaseModel):
    """Response de predicción individual."""

    churn_probability: float = Field(
        ...,
        ge=0,
        le=1,
        description="Probabilidad de churn"
    )
    prediction: Literal["churn", "no_churn"]
    confidence: float = Field(..., ge=0, le=1)
    model_version: str
    prediction_id: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)

    class Config:
        json_schema_extra = {
            "example": {
                "churn_probability": 0.73,
                "prediction": "churn",
                "confidence": 0.73,
                "model_version": "1.2.3",
                "prediction_id": "pred-abc123",
                "timestamp": "2024-01-15T10:30:00Z"
            }
        }


class BatchPredictionResponse(BaseModel):
    """Response de predicción en batch."""
    predictions: List[PredictionResponse]
    total_count: int
    processing_time_ms: float


class HealthResponse(BaseModel):
    """Response de health check."""
    status: Literal["healthy", "unhealthy"]
    model_loaded: bool
    version: str
    uptime_seconds: float


class ErrorResponse(BaseModel):
    """Response de error estándar."""
    error: str
    detail: Optional[str] = None
    request_id: Optional[str] = None
    timestamp: datetime = Field(default_factory=datetime.utcnow)
```

11.3 Service Layer (Inference)

```python
# app/services/inference.py
from pathlib import Path
from typing import List, Optional
import joblib
import pandas as pd
import numpy as np
from datetime import datetime
import uuid

from app.schemas.request import PredictionRequest
from app.schemas.response import PredictionResponse
from app.core.config import settings
from app.core.exceptions import ModelNotLoadedError, PredictionError


class InferenceService:
    """Servicio de inferencia ML."""

    def __init__(self):
        self._model = None
        self._model_version: str = "unknown"
        self._load_time: Optional[datetime] = None

    @property
    def is_ready(self) -> bool:
        """Verifica si el modelo está cargado."""
        return self._model is not None

    @property
    def model_version(self) -> str:
        return self._model_version

    @property
    def uptime_seconds(self) -> float:
        if self._load_time is None:
            return 0.0
        return (datetime.utcnow() - self._load_time).total_seconds()

    def load_model(self, model_path: Optional[Path] = None) -> None:
        """Carga el modelo desde disco."""
        path = model_path or settings.MODEL_PATH

        if not path.exists():
            raise FileNotFoundError(f"Modelo no encontrado: {path}")

        self._model = joblib.load(path)
        self._model_version = settings.MODEL_VERSION
        self._load_time = datetime.utcnow()

    def predict(self, request: PredictionRequest) -> PredictionResponse:
        """Realiza predicción individual."""
        if not self.is_ready:
            raise ModelNotLoadedError("Modelo no cargado")

        try:
            # Convertir request a DataFrame
            df = self._request_to_dataframe(request)

            # Predecir
            proba = self._model.predict_proba(df)[0, 1]
            prediction = "churn" if proba >= settings.PREDICTION_THRESHOLD else "no_churn"
            confidence = proba if proba >= 0.5 else 1 - proba

            return PredictionResponse(
                churn_probability=round(proba, 4),
                prediction=prediction,
                confidence=round(confidence, 4),
                model_version=self._model_version,
                prediction_id=f"pred-{uuid.uuid4().hex[:8]}",
            )
        except Exception as e:
            raise PredictionError(f"Error en predicción: {str(e)}")

    def predict_batch(self, requests: List[PredictionRequest]) -> List[PredictionResponse]:
        """Realiza predicciones en batch."""
        if not self.is_ready:
            raise ModelNotLoadedError("Modelo no cargado")

        # Convertir todos a DataFrame
        df = pd.concat([self._request_to_dataframe(r) for r in requests], ignore_index=True)

        # Predecir batch
        probas = self._model.predict_proba(df)[:, 1]

        responses = []
        for proba in probas:
            prediction = "churn" if proba >= settings.PREDICTION_THRESHOLD else "no_churn"
            confidence = proba if proba >= 0.5 else 1 - proba

            responses.append(PredictionResponse(
                churn_probability=round(proba, 4),
                prediction=prediction,
                confidence=round(confidence, 4),
                model_version=self._model_version,
                prediction_id=f"pred-{uuid.uuid4().hex[:8]}",
            ))

        return responses

    def _request_to_dataframe(self, request: PredictionRequest) -> pd.DataFrame:
        """Convierte PredictionRequest a DataFrame."""
        return pd.DataFrame([{
            'CreditScore': request.credit_score,
            'Age': request.age,
            'Tenure': request.tenure,
            'Balance': request.balance,
            'NumOfProducts': request.num_of_products,
            'HasCrCard': int(request.has_cr_card),
            'IsActiveMember': int(request.is_active_member),
            'EstimatedSalary': request.estimated_salary,
            'Geography': request.geography.value,
            'Gender': request.gender.value,
        }])


# Singleton para inyección de dependencias
inference_service = InferenceService()
```

11.4 Routes y Dependency Injection

```python
# app/api/deps.py
from typing import Generator
from app.services.inference import InferenceService, inference_service
from app.core.exceptions import ModelNotLoadedError

def get_inference_service() -> Generator[InferenceService, None, None]:
    """Dependency injection para InferenceService."""
    if not inference_service.is_ready:
        raise ModelNotLoadedError("Modelo no disponible")
    yield inference_service
```

```python
# app/api/routes/predict.py
from fastapi import APIRouter, Depends, HTTPException, status
from typing import List
import time

from app.schemas.request import PredictionRequest, BatchPredictionRequest
from app.schemas.response import PredictionResponse, BatchPredictionResponse
from app.services.inference import InferenceService
from app.api.deps import get_inference_service

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


@router.post(
    "",
    response_model=PredictionResponse,
    summary="Predicción individual",
    description="Predice la probabilidad de churn para un cliente.",
)
async def predict(
    request: PredictionRequest,
    service: InferenceService = Depends(get_inference_service),
) -> PredictionResponse:
    """
    Endpoint de predicción individual.

    - **credit_score**: Score crediticio (300-850)
    - **age**: Edad del cliente
    - **geography**: País (France, Germany, Spain)
    """
    return service.predict(request)


@router.post(
    "/batch",
    response_model=BatchPredictionResponse,
    summary="Predicción en batch",
    description="Predice churn para múltiples clientes (máx 1000).",
)
async def predict_batch(
    request: BatchPredictionRequest,
    service: InferenceService = Depends(get_inference_service),
) -> BatchPredictionResponse:
    """Endpoint de predicción batch."""
    start_time = time.time()

    predictions = service.predict_batch(request.instances)

    processing_time = (time.time() - start_time) * 1000

    return BatchPredictionResponse(
        predictions=predictions,
        total_count=len(predictions),
        processing_time_ms=round(processing_time, 2),
    )
```

```python
# app/api/routes/health.py
from fastapi import APIRouter, Depends
from app.schemas.response import HealthResponse
from app.services.inference import inference_service
from app.core.config import settings

router = APIRouter(tags=["Health"])


@router.get("/health", response_model=HealthResponse)
async def health_check() -> HealthResponse:
    """Health check para load balancers."""
    return HealthResponse(
        status="healthy" if inference_service.is_ready else "unhealthy",
        model_loaded=inference_service.is_ready,
        version=settings.APP_VERSION,
        uptime_seconds=inference_service.uptime_seconds,
    )


@router.get("/ready")
async def readiness_check():
    """Readiness check para Kubernetes."""
    if not inference_service.is_ready:
        raise HTTPException(status_code=503, detail="Model not loaded")
    return {"status": "ready"}
```

11.5 Middleware y Error Handling

```python
# app/middleware/logging.py
import time
import uuid
from fastapi import Request, Response
from starlette.middleware.base import BaseHTTPMiddleware
import structlog

logger = structlog.get_logger()


class LoggingMiddleware(BaseHTTPMiddleware):
    """Middleware para logging estructurado."""

    async def dispatch(self, request: Request, call_next) -> Response:
        request_id = str(uuid.uuid4())[:8]

        # Añadir request_id al state
        request.state.request_id = request_id

        start_time = time.time()

        # Log request
        logger.info(
            "request_started",
            request_id=request_id,
            method=request.method,
            path=request.url.path,
            client_ip=request.client.host if request.client else None,
        )

        response = await call_next(request)

        # Log response
        duration_ms = (time.time() - start_time) * 1000
        logger.info(
            "request_completed",
            request_id=request_id,
            status_code=response.status_code,
            duration_ms=round(duration_ms, 2),
        )

        # Añadir headers
        response.headers["X-Request-ID"] = request_id
        response.headers["X-Response-Time-Ms"] = str(round(duration_ms, 2))

        return response
```

```python
# app/core/exceptions.py
from fastapi import HTTPException, Request
from fastapi.responses import JSONResponse
from pydantic import ValidationError
import structlog

logger = structlog.get_logger()


class ModelNotLoadedError(Exception):
    """El modelo no está cargado."""
    pass


class PredictionError(Exception):
    """Error durante la predicción."""
    pass


async def model_not_loaded_handler(request: Request, exc: ModelNotLoadedError):
    """Handler para ModelNotLoadedError."""
    logger.error("model_not_loaded", path=request.url.path)
    return JSONResponse(
        status_code=503,
        content={
            "error": "Service Unavailable",
            "detail": str(exc),
            "request_id": getattr(request.state, "request_id", None),
        }
    )


async def prediction_error_handler(request: Request, exc: PredictionError):
    """Handler para PredictionError."""
    logger.error("prediction_failed", path=request.url.path, error=str(exc))
    return JSONResponse(
        status_code=500,
        content={
            "error": "Prediction Failed",
            "detail": str(exc),
            "request_id": getattr(request.state, "request_id", None),
        }
    )


async def validation_error_handler(request: Request, exc: ValidationError):
    """Handler para errores de validación Pydantic."""
    logger.warning("validation_error", errors=exc.errors())
    return JSONResponse(
        status_code=422,
        content={
            "error": "Validation Error",
            "detail": exc.errors(),
            "request_id": getattr(request.state, "request_id", None),
        }
    )
```

11.6 Main App con Lifespan

```python
# app/main.py
from contextlib import asynccontextmanager
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

from app.api.routes import health, predict
from app.middleware.logging import LoggingMiddleware
from app.core.config import settings
from app.core.exceptions import (
    ModelNotLoadedError,
    PredictionError,
    model_not_loaded_handler,
    prediction_error_handler,
)
from app.services.inference import inference_service


@asynccontextmanager
async def lifespan(app: FastAPI):
    """Lifecycle manager: startup y shutdown."""
    # Startup
    inference_service.load_model()
    yield
    # Shutdown (cleanup si necesario)


def create_app() -> FastAPI:
    """Factory function para crear la app."""

    app = FastAPI(
        title="BankChurn Predictor API",
        description="API para predicción de churn bancario",
        version=settings.APP_VERSION,
        docs_url="/docs",
        redoc_url="/redoc",
        lifespan=lifespan,
    )

    # Middleware
    app.add_middleware(
        CORSMiddleware,
        allow_origins=settings.CORS_ORIGINS,
        allow_credentials=True,
        allow_methods=["*"],
        allow_headers=["*"],
    )
    app.add_middleware(LoggingMiddleware)

    # Exception handlers
    app.add_exception_handler(ModelNotLoadedError, model_not_loaded_handler)
    app.add_exception_handler(PredictionError, prediction_error_handler)

    # Routes
    app.include_router(health.router)
    app.include_router(predict.router, prefix="/api/v1")

    return app


app = create_app()


if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

11.7 Ejercicio Integrador

Crea API Production-Ready

1. **Schemas** con validación completa
2. **Service layer** separado
3. **Dependency injection**
4. **Error handling** consistente
5. **Logging estructurado**

Checklist

```
ESTRUCTURA:
[ ] Schemas Pydantic con validación
[ ] Service layer separado de routes
[ ] Dependency injection configurada

FUNCIONALIDAD:
[ ] Endpoint /predict funcional
[ ] Endpoint /predict/batch funcional
[ ] Health check implementado

PRODUCCIÓN:
[ ] CORS configurado
[ ] Error handlers custom
[ ] Logging middleware
[ ] Request IDs
```

Siguiente Paso

Con la API lista, es hora de decidir **dónde desplegarla**.

Ir a Módulo 12: Serverless vs Contenedores →

Módulo 11 completado. Tu API ahora es production-ready.