Clase 199 — APIs con FastAPI sirviendo modelos

Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs FastAPI + Ramalho cap. 5. Duración estimada: 80 min.

Clase 199 — APIs con FastAPI sirviendo modelos

> *Parte: 4 — MLOps · Fuente: Huyen cap. 11 + docs FastAPI + Ramalho cap. 5. Duración estimada: 80 min.*

Objetivo

Exponer un modelo entrenado como REST API con FastAPI: validación de input con Pydantic, batching, async, healthcheck, métricas Prometheus, OpenAPI auto-generado. Target: p99 latency <100 ms y throughput >500 req/s en un solo nodo CPU.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Construir un servicio FastAPI con endpoints POST /predict, POST /predict-batch, GET /health, GET /metrics.
- Definir schemas de input/output con pydantic.BaseModel y obtener validación + docs gratis.
- Usar lifespan events para cargar el modelo una sola vez (no por request).
- Loadtestear con locust y medir latency p50/p95/p99 + throughput.
- Decidir entre sync def y async def según si el predict es CPU-bound o I/O-bound.

Temas

#	Tema	Por qué importa
1	ASGI vs WSGI: por qué FastAPI no es Flask	Async nativo, perf en I/O-bound.
2	Pydantic v2: validación + serialización	Schema = contrato; cambios rompen tests.
3	Lifespan: cargar modelo 1 vez	Sin esto, cada request reabre joblib.load.
4	Sync vs async para predict	CPU-bound → sync (deja al thread pool); I/
5	Batching: /predict-batch	100 predicciones en 1 request, no en 100.
6	Observabilidad: /health, /metrics, logs es	Lo que pide el oncall a las 3 AM.

Definiciones y características

- FastAPI: framework ASGI (no WSGI). Web server: uvicorn (single-process) o gunicorn -w N -k uvicorn.workers.UvicornWorker (multi-worker).
- ASGI (Asynchronous Server Gateway Interface): protocolo Python para apps async. Permite async def endpoints, websockets, streaming.
- Pydantic v2: validación + serialización con modelos tipados. ~20× más rápido que v1 (re-escrito en Rust).
- lifespan event (@asynccontextmanager): hook para inicializar recursos (cargar modelo, abrir DB) cuando el server arranca y limpiarlos al apagar. Reemplaza el deprecado @app.on_event("startup").
- Sync vs async endpoint: en FastAPI, def predict(...) (sync) corre en un thread pool — bueno para CPU-bound o libs que no son async. async def predict(...) corre en el event loop — bueno para I/O-bound (DB, HTTP a otro servicio). Si tu predict es solo model.predict(X) (CPU): sync.

- Batching: agrupar N predicciones en un solo request. Reduce overhead de HTTP (~5 ms por request) y permite vectorizar (model.predict(matrix) es ~10× más rápido que 100 model.predict(vector)).
- Healthcheck: endpoint que devuelve 200 si el servicio puede servir tráfico. K8s lo usa para livenessProbe (reiniciar pod si falla) y readinessProbe (sacar del LB si falla).

Dataset / recursos

- Modelo: cualquiera entrenado en clases previas — usamos sklearn por simplicidad.
- Librerías: fastapi, uvicorn[standard], pydantic>=2, prometheus-fastapi-instrumentator, locust (loadtest).

Ejercicios

1. API mínima: POST /predict con IrisInput(features: list[float]) → IrisOutput(class: int, proba: list[float]). Levantá con uvicorn app:app --reload. Abrí /docs (OpenAPI Swagger UI). Confirmá que probar desde la UI funciona.
2. Lifespan: cargá el modelo en un lifespan y guardalo en app.state.model. Verificá que el modelo se carga UNA vez (print al inicio) aunque hagas 100 requests.
3. Batching: agregá POST /predict-batch con BatchInput(rows: list[list[float]]). Medí latencia de 100 predicciones individuales vs 1 batch de 100.
4. Async vs sync: simulá un predict que llama a una API externa (await httpx.AsyncClient().get(...)). Compará def (bloquea thread pool) vs async def (libera event loop). Loadtest con 200 concurrent users.
5. Observabilidad: agregá prometheus-fastapi-instrumentator → /metrics. Healthcheck /health que devuelve {"status": "ok", "model_loaded": bool}. Loggeá cada request con logger.info (JSON).

Homework verificable

Servicio FastAPI + Dockerfile (Clase 198) con:

1. Endpoints POST /predict, POST /predict-batch, GET /health, GET /metrics, GET /docs.
2. Pydantic models con validación: features debe tener exactamente 4 floats positivos.
3. lifespan que carga model.pkl una vez al startup.
4. locustfile.py que simula 100 users concurrentes durante 60 s.
5. Reporte de loadtest con p50/p95/p99 latency y RPS, justificando si el target (<100 ms p99, >500 RPS) se cumple.

Criterio de aceptación: curl -X POST localhost:8000/predict -d '{"features":[-1,2,3,4]}' -H 'content-type:application/json' devuelve HTTP 422 (validación rechaza valor negativo). El loadtest cumple p99 <100 ms en al menos un workload.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Latency altísima (p99 >500 ms) con un mode	Estás cargando joblib.load("model.pkl") de
Error RuntimeError: This event loop is alr	Mezclaste asyncio.run(...) dentro de un en
422 Unprocessable Entity en requests que p	Pydantic v2 es estricto: list[float] recha
Single worker satura 1 CPU	uvicorn app:app corre 1 worker. Fix: gunic
/docs está activo en producción	Por default FastAPI expone /docs, /redoc,

Healthcheck devuelve 200 aunque el modelo	Healthcheck simplista. Fix: chequear app.s

Preguntas frecuentes

¿FastAPI o Flask?

Para servir modelos hoy: FastAPI. Pros: async nativo, validación con Pydantic (no request.json + checks manuales), OpenAPI gratis, performance superior en benchmarks (~3× Flask). Flask sigue siendo elegible para apps simples o equipos con experiencia previa.

¿Cuándo async def y cuándo def?

Regla: si el endpoint solo hace CPU-bound (cargar modelo, predecir): def (FastAPI lo manda al thread pool, no bloquea el event loop). Si hace I/O-bound (await DB, await HTTP): async def. Si mezclás requests (sync HTTP) en un async def: bloqueás el loop. Cambialo a httpx.AsyncClient.

¿Servir con uvicorn o gunicorn?

Dev: uvicorn --reload. Prod: gunicorn -w N -k uvicorn.workers.UvicornWorker (N = 2 * cores + 1). Razón: gunicorn maneja restarts, multi-proceso, signals. En K8s con HPA + 1 worker/pod: uvicorn solo está bien.

¿Cómo manejo modelos grandes (>1 GB) que tardan en cargar?

(1) lifespan (no re-cargar). (2) readinessProbe con initialDelaySeconds: 60 — pod no recibe tráfico hasta cargar. (3) model.eval() + torch.jit.script (PyTorch) o onnxruntime (cualquier framework) — más rápido en inferencia que el framework original. (4) Para LLMs: vLLM/TGI con paged attention (Clase 165).

¿Validación strict de Pydantic me rompe clientes legacy?

Sí, si pasaban tipos laxos. Pydantic v2 puede ser permisivo con model_config = {"strict": False} o usar validators (field_validator). Lo correcto a mediano plazo: docs claras + versioning de API (/v1/predict vs /v2/predict).

¿FastAPI sirve modelos PyTorch/TF directamente?

Sí, pero para producción seria conviene un inference server dedicado: TorchServe, TensorFlow Serving, NVIDIA Triton (multi-framework, batching dinámico, GPU optimal). FastAPI queda como gateway/proxy con auth y reglas de negocio. Para casos simples sklearn/XGBoost: FastAPI alcanza.

Referencias

- Huyen, Chip. Designing Machine Learning Systems (O'Reilly, 2022), cap. 11.
- FastAPI docs — Tutorial y Advanced User Guide.
- Pydantic v2 migration guide — qué cambió desde v1.
- Locust — loadtest distribuido en Python.
- NVIDIA Triton — alternativa para producción seria.

Siguiente clase

Clase 200 — Kubernetes para servir modelos a escala

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import os, shutil, tempfile, joblib
from pathlib import Path
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

WORK = Path(tempfile.gettempdir()) / 'fastapi_demo'
if WORK.exists(): shutil.rmtree(WORK)
WORK.mkdir(); os.chdir(WORK)

X, y = load_iris(return_X_y=True)
m = RandomForestClassifier(n_estimators=50, random_state=42).fit(X, y)
joblib.dump(m, 'model.pkl')
print('modelo guardado:', Path('model.pkl').stat().st_size, 'bytes')
```

Archivos complementarios

- notebook.ipynb