Week 10: HTTP APIs & FastAPI

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Recap: The Web Data Pipeline

Week 1: We were **Consumers**.

- We *sent* HTTP requests (`GET`) to fetch data.
- Tools: `curl` , `requests` , Browser.

Week 10 (Today): We become **Producers**.

- We will *build* the server that accepts requests.
- We will *serve* our ML models to the world.
- Tool: **FastAPI**.

Why Build APIs for AI?

1. **The "Inference" Gap**
    - You have a trained model in a Jupyter Notebook (`model.pkl`).
    - A mobile app developer wants to use it.
    - They can't run your notebook!
2. **Solution: The API Contract**
    - The app sends a JSON request: `{"features": [1.5, 2.0]}`
    - Your API runs the model and returns JSON: `{"prediction": "Cat"}`
    - **Decoupling**: The app doesn't care if you use PyTorch, sklearn, or magic.

Why FastAPI?

Modern Python web framework for APIs

Key Features:

- **Fast**: High performance (on par with NodeJS/Go).
- **Type Hints**: Uses Python types for validation.
- **Auto-Docs**: Generates Swagger UI automatically.
- **Async**: Native support for concurrency.
- **Standards**: Based on JSON Schema and OpenAPI.

Alternatives: Flask (older, simpler), Django (full-stack, heavy).

FastAPI Setup

Installation:

```
pip install "fastapi[standard]"
```

Hello World (`main.py`):

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello World"}
```

Run server:

```
fastapi dev main.py
```

The Request-Response Cycle (Server Side)

1. **Listen**: Server waits on a port (e.g., 8000).
2. **Route**: URL `/predict` matches a specific Python function.
3. **Validate**: Check if input data matches expected format (Pydantic).
4. **Process**: Run your logic (inference).
5. **Response**: Convert result to JSON and send back.

Path Parameters

Dynamic URL segments:

```python
@app.get("/items/{item_id}")
def read_item(item_id: int):
    # FastAPI automatically converts string "42" to int 42
    return {"item_id": item_id}
```

Validation is automatic:

- Request: `GET /items/42` -> 200 OK
- Request: `GET /items/foo` -> 422 Unprocessable Entity (Validation Error)

Query Parameters

Key-value pairs after `?`:

```python
# URL: /items/?skip=0&limit=10
@app.get("/items/")
def read_items(skip: int = 0, limit: int = 10):
    return fake_items_db[skip : skip + limit]
```

Optional parameters:

```python
from typing import Optional

@app.get("/search")
def search(q: Optional[str] = None):
    if q:
        return {"results": perform_search(q)}
    return {"results": []}
```

Request Body: The Pydantic Power

Sending data (POST) requires a schema.

```python
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = None

@app.post("/items/")
def create_item(item: Item):
    return {"item_name": item.name, "item_price": item.price}
```

Client sends JSON:

```json
{"name": "Hammer", "price": 15.5}
```

FastAPI parses it into a Python object

Pydantic Validation for ML

Enforce constraints on your model inputs:

```python
from pydantic import BaseModel, Field

class PredictionRequest(BaseModel):
    # Ensure age is realistic
    age: int = Field(gt=0, lt=120)

    # Ensure income is positive
    income: float = Field(gt=0)

    # Categorical validation? Use Enums!
```

Why this matters:

- Prevents garbage data from crashing your model.
- Provides clear error messages to API users.

Serving an ML Model

Pattern: Load Global, Predict Local

```python
import joblib
from fastapi import FastAPI

app = FastAPI()
model = None

# Load model ONCE when app starts
@app.on_event("startup")
def load_model():
    global model
    model = joblib.load("iris_model.pkl")

@app.post("/predict")
def predict(features: list[float]):
    prediction = model.predict([features])
    return {"class": int(prediction[0])}
```

Handling Files (Images/Audio)

For Computer Vision or Audio models:

```python
from fastapi import File, UploadFile
from PIL import Image
import io

@app.post("/classify-image")
async def classify(file: UploadFile = File(...)):
    # Read bytes
    contents = await file.read()

    # Convert to PIL Image
    image = Image.open(io.BytesIO(contents))

    # Run inference...
    return {"label": "cat", "confidence": 0.98}
```

Automatic Documentation (Swagger UI)

The Killer Feature.

Navigate to `http://localhost:8000/docs`.

You get an interactive UI to:

- See all endpoints.
- See required JSON schemas.
- **Try it out** button to send requests directly from the browser.

No more writing API docs manually!

Async/Await: Concurrency

Python is synchronous by default.

- If one request takes 5s, everyone waits.

Async allows concurrency:

- While waiting for DB/Disk/Network, handle other requests.

```python
@app.get("/")
async def read_root():
    # await database_query()
    return {"message": "I am non-blocking!"}
```

Rule of Thumb for ML:

- `model.predict()` is CPU-bound (blocking).
- Define ML endpoints with standard `def` (FastAPI runs them in a thread pool).

Testing APIs

Use `TestClient` (based on requests/httpx).

```python
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)

def test_predict():
    response = client.post(
        "/predict",
        json={"features": [5.1, 3.5, 1.4, 0.2]}
    )
    assert response.status_code == 200
    assert response.json() == {"class": 0}
```

Run with `pytest`.

Dependency Injection

Reusable logic (Database sessions, Auth tokens).

```
from fastapi import Depends

def get_token_header(x_token: str = Header()):
    if x_token != "secret-token":
        raise HTTPException(status_code=400, detail="Invalid Header")

@app.get("/items/", dependencies=[Depends(get_token_header)])
def read_items():
    return [{"item": "Foo"}, {"item": "Bar"}]
```

Error Handling in FastAPI

Custom exceptions for better error messages:

```python
from fastapi import HTTPException

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id not in database:
        raise HTTPException(
            status_code=404,
            detail=f"Item {item_id} not found",
            headers={"X-Error": "ItemNotFound"}
        )
    return database[item_id]
```

Custom exception handlers:

```python
from fastapi.responses import JSONResponse
```

Input Validation Best Practices

Use Pydantic validators for complex rules:

```python
from pydantic import BaseModel, validator, Field

class PredictionInput(BaseModel):
    age: int = Field(gt=0, lt=120, description="Age in years")
    income: float = Field(gt=0)
    credit_score: int = Field(ge=300, le=850)

    @validator('income')
    def income_realistic(cls, v):
        if v > 10_000_000:  # $10M annual income
            raise ValueError('Income seems unrealistic')
        return v

    @validator('credit_score')
    def validate_credit(cls, v):
        if v < 300 or v > 850:
            raise ValueError('Credit score out of valid range')
```

Response Models

Control what data is returned:

```python
class User(BaseModel):
    username: str
    email: str
    password: str  # Sensitive!

class UserResponse(BaseModel):
    username: str
    email: str
    # Password excluded

@app.post("/users/", response_model=UserResponse)
def create_user(user: User):
    # Save user with password
    save_to_db(user)
    # Return without password
    return user
```

API Versioning Strategies

Why version?

- Maintain backward compatibility
- Allow gradual migrations
- Support multiple clients

URL versioning:

```
@app.post("/v1/predict")
def predict_v1(data: InputV1):
    return old_model.predict(data)

@app.post("/v2/predict")
def predict_v2(data: InputV2):
    return new_model.predict(data)
```

Header versioning:

CORS (Cross-Origin Resource Sharing)

Problem: Browsers block requests from different domains by default.

Solution: Enable CORS for web apps to call your API.

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # In production: specific domains
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Production: Replace `["*"]` with specific origins:

```
allow_origins=["https://myapp.com", "https://dashboard.myapp.com"]
```

Rate Limiting

Protect your API from abuse:

```python
from slowapi import Limiter, _rate_limit_exceeded_handler
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)

@app.get("/predict")
@limiter.limit("10/minute")  # Max 10 requests per minute per IP
async def predict(request: Request):
    return {"prediction": model.predict(input)}
```

Why needed:

- Prevent DoS attacks
- Manage GPU/CPU usage

Logging and Monitoring

Structured logging for production:

```
import logging
from datetime import datetime

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@app.post("/predict")
def predict(input_data: PredictionInput):
    start_time = datetime.now()

    logger.info(f"Prediction request received: {input_data}")

    try:
        result = model.predict(input_data)

        latency = (datetime.now() - start_time).total_seconds()
        logger.info(f"Prediction successful, latency: {latency}s")

        return {"prediction": result}

    except Exception as e:
```

Health Check Endpoint

Essential for deployment and monitoring:

```
@app.get("/health")
def health_check():
    return {
        "status": "healthy",
        "model_loaded": model is not None,
        "version": "1.0.0",
        "timestamp": datetime.now().isoformat()
    }

@app.get("/readiness")
def readiness_check():
    # More thorough check
    if model is None:
        raise HTTPException(status_code=503, detail="Model not loaded")

    # Test model
    try:
        test_input = [[1.0, 2.0, 3.0, 4.0]]
        _ = model.predict(test_input)
        return {"status": "ready"}
```

Security Best Practices

1. API Key Authentication:

```
from fastapi.security import APIKeyHeader

API_KEY = "your-secret-key"
api_key_header = APIKeyHeader(name="X-API-Key")

def verify_api_key(api_key: str = Depends(api_key_header)):
    if api_key != API_KEY:
        raise HTTPException(status_code=403, detail="Invalid API Key")

@app.post("/predict", dependencies=[Depends(verify_api_key)])
def predict(data: Input):
    return model.predict(data)
```

2. Input sanitization:

- Validate all inputs with Pydantic

Request/Response Examples

Document API with examples:

```
class PredictionInput(BaseModel):
    features: List[float] = Field(
        ...,
        example=[5.1, 3.5, 1.4, 0.2],
        description="Iris measurements"
    )

    class Config:
        schema_extra = {
            "example": {
                "features": [5.1, 3.5, 1.4, 0.2]
            }
        }

@app.post("/predict", response_model=PredictionOutput)
def predict(input: PredictionInput):
    """
    Predict iris species from measurements.

    - **features**: List of 4 floats [sepal_length, sepal_width, petal_length, petal_width]
    - **Returns**: Predicted species and confidence
```

Background Tasks

For non-blocking operations:

```python
from fastapi import BackgroundTasks

def log_prediction(input_data: dict, prediction: dict):
    # Save to database
    db.save_prediction(input_data, prediction)

@app.post("/predict")
def predict(input: Input, background_tasks: BackgroundTasks):
    result = model.predict(input)

    # Log asynchronously
    background_tasks.add_task(log_prediction, input.dict(), result)

    return result
```

Use cases: Logging, sending emails, updating analytics.

File Uploads (Images/Audio)

Handling multipart form data:

```
from fastapi import File, UploadFile
import numpy as np
from PIL import Image
import io

@app.post("/classify-image")
async def classify_image(file: UploadFile = File(...)):
    # Validate file type
    if file.content_type not in ["image/jpeg", "image/png"]:
        raise HTTPException(400, "Invalid file type")

    # Read and process
    contents = await file.read()
    image = Image.open(io.BytesIO(contents))

    # Preprocess
    image = image.resize((224, 224))
    img_array = np.array(image) / 255.0

    # Predict
    prediction = model.predict(np.expand_dims(img_array, 0))

    return {
        "filename": file.filename,
        "class": int(prediction[0]),
```

Streaming Responses

For large datasets or LLM outputs:

```python
from fastapi.responses import StreamingResponse
import time

def generate_text():
    prompt = "Once upon a time"
    for token in llm.generate(prompt):
        yield f"data: {token}\n\n"
        time.sleep(0.1)

@app.get("/generate")
async def stream():
    return StreamingResponse(
        generate_text(),
        media_type="text/event-stream"
    )
```

Client receives tokens as they're generated (Server-Sent Events)

Deployment Considerations

Production checklist:

1. **Logging**: Structured logs (JSON format)
2. **Error handling**: Catch all exceptions
3. **Validation**: Strict Pydantic models
4. **Security**: API keys, rate limiting, HTTPS
5. **Monitoring**: Health checks, metrics
6. **Documentation**: Auto-generated + custom
7. **Testing**: Unit tests, integration tests
8. **Performance**: Caching, async where possible
9. **Versioning**: API versioning strategy
10. **CORS**: Configured for your frontend

Production Servers

Development server (`fastapi dev`):

- Auto-reload on code changes
- Detailed error messages
- **NOT for production**

Production servers:

Uvicorn (ASGI server):

```
uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4
```

Gunicorn + Uvicorn workers (better):

```
gunicorn main:app --workers 4 --worker-class uvicorn.workers.UvicornWorker
```

Best Practices for ML APIs

1. **Batching**: Batch requests for GPU efficiency
2. **Validation**: Strict Pydantic models
3. **Versioning**: `/v1/predict`, `/v2/predict`
4. **Logging**: Log inputs for drift detection
5. **Health checks**: `/health` and `/readiness`
6. **Error handling**: Graceful failures with clear messages
7. **Documentation**: Use Pydantic examples
8. **Testing**: Comprehensive test coverage
9. **Security**: API keys, rate limiting
10. **Monitoring**: Track latency, throughput, errors

Performance Optimization

Model loading:

- Load once at startup (global variable)
- Use `@app.on_event("startup")`

Caching:

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def expensive_preprocessing(input_text: str):
    return process(input_text)
```

Async for I/O-bound tasks:

```python
@app.get("/data")
async def get_data():
```

API Documentation Best Practices

Auto-generated docs (Swagger UI):

- Available at `/docs`
- Interactive testing
- Shows request/response schemas

ReDoc (alternative view):

- Available at `/redoc`
- More readable for humans

Custom descriptions:

```
app = FastAPI(
    title="ML Prediction API",
    description="API for serving machine learning models",
```

Comparison: FastAPI vs Alternatives

Feature	FastAPI	Flask	Django REST
Performance	Very High	Medium	Medium
Async Support	Native	Partial	Limited
Type Validation	Automatic	Manual	Manual
Auto Docs	Yes	No	Partial
Learning Curve	Easy	Easy	Steep
Best For	ML APIs, Modern apps	Simple apps	Full stack

FastAPI is ideal for ML model serving.

RESTful API Design Principles

REST (Representational State Transfer): Architectural style for APIs

Key principles:

1. **Resource-based URLs**:

```
GET    /models                # List all models
GET    /models/{id}           # Get specific model
POST   /models                # Create new model
PUT    /models/{id}           # Update model
DELETE /models/{id}           # Delete model
```

2. **HTTP methods are semantic**:

- GET: Read (idempotent, no side effects)
- POST: Create (not idempotent)
- PUT: Update/replace (idempotent)

API Versioning Strategies

Problem: API changes break existing clients.

Strategy 1: URL versioning:

```
@app.post("/v1/predict")  # Old version
async def predict_v1(data: InputV1):
    return model_v1.predict(data)

@app.post("/v2/predict")  # New version
async def predict_v2(data: InputV2):
    return model_v2.predict(data)
```

Strategy 2: Header versioning:

```
from fastapi import Header

@app.post("/predict")
async def predict(data: Input, api_version: str = Header("1.0")):
```

Middleware and Dependency Injection

Middleware: Code that runs before/after requests.

```
@app.middleware("http")
async def add_timing_header(request: Request, call_next):
    """Add response time header."""
    start_time = time.time()

    response = await call_next(request)

    process_time = time.time() - start_time
    response.headers["X-Process-Time"] = str(process_time)

    return response
```

Dependency Injection: Reusable components.

```
from fastapi import Depends
```

Caching Strategies for ML APIs

Problem: Model inference is expensive, many duplicate requests.

Solution: Cache predictions.

In-memory caching (simple):

```
from functools import lru_cache

@lru_cache(maxsize=1000)
def predict_cached(input_hash: str):
    """Cache predictions by input hash."""
    return model.predict(input_data)

@app.post("/predict")
async def predict(data: Input):
    # Hash input for cache key
    input_hash = hashlib.md5(str(data).encode()).hexdigest()

    result = predict_cached(input_hash)
```

Load Balancing and Horizontal Scaling

Problem: Single server can't handle all traffic.

Solution: Run multiple instances, distribute load.

Load balancer (Nginx):

```
upstream api_servers {
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
    server 127.0.0.1:8002;
}

server {
    listen 80;
    location / {
        proxy_pass http://api_servers;
    }
}
```

Rate Limiting Algorithms

Token Bucket: Classic algorithm.

Algorithm:

1. Bucket starts with N tokens
2. Each request consumes 1 token
3. Tokens refill at rate R per second
4. If bucket empty, reject request

```
import time

class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate  # tokens per second
        self.capacity = capacity
        self.tokens = capacity
        self.last_update = time.time()

    def consume(self, tokens=1):
        """Try to consume tokens. Return True if allowed."""
        now = time.time()
```

Authentication and Authorization

Authentication: Who are you?

Authorization: What can you do?

API Key authentication (simple):

```
API_KEYS = {"abc123": "user1", "def456": "user2"}

def verify_api_key(api_key: str = Header(...)):
    """Verify API key."""
    if api_key not in API_KEYS:
        raise HTTPException(401, "Invalid API key")
    return API_KEYS[api_key]

@app.post("/predict")
async def predict(data: Input, user: str = Depends(verify_api_key)):
    log_prediction(user, data)
    return model.predict(data)
```

WebSockets for Real-Time ML

Problem: HTTP is request-response. What about streaming predictions?

Solution: WebSockets (bidirectional, persistent connection).

```
from fastapi import WebSocket

@app.websocket("/ws/predict")
async def websocket_predict(websocket: WebSocket):
    """Stream predictions over WebSocket."""
    await websocket.accept()

    try:
        while True:
            # Receive data from client
            data = await websocket.receive_text()

            # Run inference
            prediction = model.predict(data)

            # Send result back
            await websocket.send_json({"prediction": prediction})

    except WebSocketDisconnect:
        print("Client disconnected")
```

API Gateway Pattern

Problem: Multiple microservices, client needs to call all of them.

Solution: API Gateway (single entry point).

Architecture:

```
Client → API Gateway → Model A
                    → Model B
                    → Database
```

Benefits:

- Single authentication point
- Rate limiting across services
- Request routing and aggregation
- Caching and compression

Summary

Key concepts:

1. FastAPI = Modern, fast, type-safe Python web framework
2. Pydantic = Automatic validation and documentation
3. Async = Handle concurrent requests efficiently
4. Testing = TestClient for automated verification
5. Production = Health checks, logging, security

ML-specific patterns:

- Load model once at startup
- Validate inputs strictly
- Log all predictions

Lab Preview

Today you will:

1. Build Hello World API
2. Add input validation with Pydantic
3. **Serve a Scikit-Learn model**
4. Implement error handling
5. Add health check endpoint
6. Write comprehensive tests
7. Deploy locally with production server

Let's build production-ready ML APIs!