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 `?`:

```
# 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:

```
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.

```
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:

```
{"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:

```
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).

```
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"}]
```

Best Practices for ML APIs

1. **Batching**: If high traffic, batch requests before sending to GPU.
2. **Validation**: Strict Pydantic models preventing bad inputs.
3. **Versioning**: `/v1/predict`, `/v2/predict`.
4. **Logging**: Log inputs (for drift detection) and latencies.
5. **Health Check**: `/health` endpoint for monitoring tools.

Lab Preview

Today you will:

1. Build a "Unit Converter" API (warmup).
2. **Serve a Scikit-Learn Model**:
    - Train a simple model.
    - Save it (`joblib`).
    - Build a `/predict` endpoint.
3. **Input Validation**: Ensure robust inputs.
4. **Test**: Write a test case for your API.

Let's build!