

# Week 10: Building ML APIs with FastAPI

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

# The Problem: Sharing Your Model

**You've built an amazing movie predictor:**

- Trained a model that predicts Netflix success
- Works great in your Jupyter notebook

**But now:**

- Your professor wants to try it
- A mobile app developer wants to use it
- The marketing team wants a dashboard

**Challenge:** They can't run your notebook!

# The Solution: APIs

## API = Application Programming Interface

Think of it like a restaurant:

- You (client) order food through a menu (API)
- Kitchen (server) prepares the food
- You don't need to know HOW the kitchen works

## For ML:

Request	Response
Send movie features	Get prediction back
<code>{"budget": 100, "genre": "Action"}</code>	<code>{"success": true, "confidence": 0.85}</code>

# HTTP: The Language of the Web

Every web request has:

Method	Purpose	Example
GET	Read data	Get user profile
POST	Send data	Submit prediction
PUT	Update data	Update settings
DELETE	Remove data	Delete account

For ML predictions: We mostly use **POST** (send data, get prediction)

# What is FastAPI?

**FastAPI** = Python framework for building APIs

## Why FastAPI?

1. **Fast**: High performance (like Node.js)
2. **Easy**: Write Python, get web APIs
3. **Auto-docs**: Interactive documentation for free
4. **Validation**: Automatic input checking

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

# Your First API: Hello World

Create `app.py` :

```
from fastapi import FastAPI

app = FastAPI()

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

Run it:

```
fastapi dev app.py
```

Visit: <http://localhost:8000>

# Understanding the Code

```
from fastapi import FastAPI

app = FastAPI() # Create the app

@app.get("/") # When someone visits "/"
def hello(): # Run this function
    return {"message": "Hello, World!"} # Return JSON
```

The `@app.get("/")` decorator:

- `@app.get` = Handle GET requests
- `"/"` = At the root URL (<http://localhost:8000/>)

# Auto-Generated Documentation

**Visit:** <http://localhost:8000/docs>

You get **Swagger UI** for free:

- See all your endpoints
- Test them interactively
- View request/response schemas

**Try it:** Click "Try it out" and execute!



# Path Parameters

## Dynamic URLs:

```
@app.get("/movies/{movie_id}")  
def get_movie(movie_id: int):  
    return {"movie_id": movie_id, "title": "Movie " + str(movie_id)}
```

## Examples:

- `/movies/42` → `{"movie_id": 42, "title": "Movie 42"}`
- `/movies/123` → `{"movie_id": 123, "title": "Movie 123"}`

**Note:** FastAPI automatically converts `"42"` to integer `42`

# Query Parameters

## Optional parameters in URL:

```
@app.get("/movies")
def search_movies(genre: str = None, limit: int = 10):
    return {
        "genre": genre,
        "limit": limit,
        "results": ["Movie 1", "Movie 2"]
    }
```

## Examples:

- `/movies` → Uses defaults
- `/movies?genre=Action` → Filter by genre
- `/movies?genre=Comedy&limit=5` → Both parameters

# POST Requests: Sending Data

For predictions, we need to send data:

```
from pydantic import BaseModel

class MovieInput(BaseModel):
    genre: str
    budget: float
    runtime: int

@app.post("/predict")
def predict(movie: MovieInput):
    # Prediction logic here
    return {"success": True, "confidence": 0.85}
```

Client sends JSON:

```
{"genre": "Action", "budget": 150.0, "runtime": 120}
```

# What is Pydantic?

Pydantic = Data validation library

```
from pydantic import BaseModel

class MovieInput(BaseModel):
    genre: str      # Must be a string
    budget: float   # Must be a number
    runtime: int    # Must be an integer
```

FastAPI uses Pydantic to:

1. Validate incoming data
2. Show schema in docs
3. Return clear error messages

# Pydantic Field Validation

Add constraints:

```
from pydantic import BaseModel, Field

class MovieInput(BaseModel):
    genre: str
    budget: float = Field(gt=0, lt=500) # Between 0-500
    runtime: int = Field(ge=60, le=240) # 60-240 minutes
    is_sequel: bool = False # Optional with default
```

If budget is -10:

```
{"detail": "budget must be greater than 0"}
```

# Serving Your ML Model

## The pattern:

```
import joblib
from fastapi import FastAPI

app = FastAPI()
model = None # Global variable

@app.on_event("startup")
def load_model():
    global model
    model = joblib.load("movie_model.pkl")
    print("Model loaded!")
```

## Why load at startup?

- Load once, use many times
- Don't reload for each request

# Complete Prediction Endpoint

```
@app.post("/predict")
def predict(movie: MovieInput):
    # Prepare features
    features = [
        movie.budget,
        movie.runtime,
        1 if movie.is_sequel else 0
    ]

    # Make prediction
    prediction = model.predict([features])[0]
    probability = model.predict_proba([features])[0]

    return {
        "prediction": "Success" if prediction == 1 else "Risky",
        "confidence": float(max(probability))
    }
```

# Complete Example: Movie Predictor API

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
import joblib

app = FastAPI(title="Movie Success Predictor")
model = None

class MovieInput(BaseModel):
    budget: float = Field(gt=0, description="Budget in millions")
    runtime: int = Field(ge=60, le=240, description="Runtime in minutes")
    is_sequel: bool = False

class PredictionOutput(BaseModel):
    prediction: str
    confidence: float

@app.on_event("startup")
def load_model():
    global model
    model = joblib.load("movie_model.pkl")
```



# Complete Example (continued)

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

@app.post("/predict", response_model=PredictionOutput)
def predict(movie: MovieInput):
    if model is None:
        raise HTTPException(status_code=503, detail="Model not loaded")

    features = [[movie.budget, movie.runtime, int(movie.is_sequel)]]
    prediction = model.predict(features)[0]
    probability = model.predict_proba(features)[0]

    return {
        "prediction": "Success" if prediction == 1 else "Risky",
        "confidence": float(max(probability))
    }
```

# Error Handling

What if something goes wrong?

```
from fastapi import HTTPException

@app.post("/predict")
def predict(movie: MovieInput):
    if model is None:
        raise HTTPException(
            status_code=503,
            detail="Model not available"
        )

    try:
        result = model.predict(...)
        return {"prediction": result}
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"Prediction failed: {str(e)}"
        )
```

# HTTP Status Codes

Code	Meaning	When to use
200	OK	Request succeeded
400	Bad Request	Invalid input
404	Not Found	Resource doesn't exist
422	Validation Error	Pydantic validation failed
500	Server Error	Something crashed
503	Unavailable	Model not loaded

# Health Check Endpoint

Always add a health check:

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

Why?

- Kubernetes/Docker checks if app is running
- Load balancers know where to send traffic
- Monitoring tools track uptime

# Testing Your API

## Use FastAPI's TestClient:

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

client = TestClient(app)

def test_health():
    response = client.get("/health")
    assert response.status_code == 200
    assert response.json()["status"] == "healthy"

def test_predict():
    response = client.post("/predict", json={
        "budget": 100, "runtime": 120, "is_sequel": False
    })
    assert response.status_code == 200
    assert "prediction" in response.json()
```

# Running Tests

Install pytest:

```
pip install pytest
```

Create `test_app.py` with your tests

Run:

```
pytest test_app.py -v
```

Output:

```
test_app.py::test_health PASSED  
test_app.py::test_predict PASSED
```

# Running in Production

Development server (for testing):

```
fastapi dev app.py # Auto-reload on changes
```

Production server:

```
uvicorn app:app --host 0.0.0.0 --port 8000
```

With multiple workers:

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

# Docker Deployment

Create Dockerfile :

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Build and run:

```
docker build -t movie-api .
docker run -p 8000:8000 movie-api
```



# Calling Your API from Python

Using requests library:

```
import requests

# Make prediction
response = requests.post(
    "http://localhost:8000/predict",
    json={
        "budget": 100,
        "runtime": 120,
        "is_sequel": False
    }
)

print(response.json())
# {"prediction": "Success", "confidence": 0.85}
```

# Calling Your API from JavaScript

```
// In a web app
fetch("http://localhost:8000/predict", {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify({
    budget: 100,
    runtime: 120,
    is_sequel: false
  })
})
.then(response => response.json())
.then(data => console.log(data));
```

# CORS: Allowing Web Apps to Call Your API

**Problem:** Browsers block requests to different domains

**Solution:** Enable CORS

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"], # Allow all (use specific domains in production)
    allow_methods=["*"],
    allow_headers=["*"],
)
```

# API Best Practices

## 1. Always validate inputs

```
budget: float = Field(gt=0, lt=500)
```

## 2. Return consistent responses

```
{"prediction": ..., "confidence": ...}
```

## 3. Add health checks

```
@app.get("/health")
```

## 4. Handle errors gracefully

```
except Exception as e:  
    raise HTTPException(500, detail=str(e))
```

## 5. Document your API (FastAPI does this automatically!)

# Summary

Concept	What it does
<b>FastAPI</b>	Python framework for APIs
<b>Pydantic</b>	Data validation
<b>GET</b>	Read data
<b>POST</b>	Send data (predictions)
<b>HTTPException</b>	Handle errors
<b>/health</b>	Check if API is running
<b>TestClient</b>	Test your API

# Lab Preview

## This week you'll:

1. Create a FastAPI app from scratch
2. Add input validation with Pydantic
3. Serve your Netflix movie predictor model
4. Add error handling and health checks
5. Write tests for your API
6. Deploy with Docker (optional)

**Result:** A working ML API you can share with anyone!

# Questions?

## Key takeaways:

- FastAPI makes building ML APIs easy
- Pydantic validates your inputs automatically
- Always add health checks and error handling
- Test your API before deployment

**Next week:** Git and CI/CD