Week 10 Lab: Building ML APIs with FastAPI

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Lab Objectives

By the end of this lab, you will:

1. **Build REST APIs** with FastAPI from scratch
2. **Validate inputs** using Pydantic models
3. **Serve ML models** via HTTP endpoints
4. **Handle errors** gracefully
5. **Test APIs** with automated tests
6. **Deploy** with production servers

Prerequisites:

```
pip install "fastapi[standard]" scikit-learn joblib pytest httpx
```

Part 1: FastAPI Fundamentals

Getting started with API development

Exercise 1.1: Hello FastAPI (15 min)

Goal: Create your first API and explore Swagger UI

Create `main.py`:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI!"}

@app.get("/hello/{name}")
def greet(name: str):
    return {"greeting": f"Hello, {name}!"}
```

Run:

Exercise 1.1: Explore Swagger UI

Try these in the Swagger UI:

1. Click on `GET /` → "Try it out" → "Execute"
2. Click on `GET /hello/{name}` → Enter your name → "Execute"

Observations:

- Interactive documentation automatically generated
- Can test all endpoints from the browser
- Shows request/response schemas

Try via curl:

```
curl http://127.0.0.1:8000/
curl http://127.0.0.1:8000/hello/Alice
```

Exercise 1.2: Path and Query Parameters (20 min)

Goal: Understand different ways to pass data

Add to `main.py`:

```
from typing import Optional

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    result = {"item_id": item_id}
    if q:
        result["query"] = q
    return result

@app.get("/search")
def search(
    keyword: str,
    category: Optional[str] = None,
    limit: int = 10
):
    return {
```

Exercise 1.2: Test Your Endpoints

Try these URLs:

- http://127.0.0.1:8000/items/42
- http://127.0.0.1:8000/items/42?q=test
- http://127.0.0.1:8000/search?keyword=python
- http://127.0.0.1:8000/search?keyword=python&category=books&limit=5

What happens if:

- You use `/items/abc` instead of a number?
- You omit the required `keyword` parameter?

Answer: FastAPI automatically validates and returns 422 error!

Exercise 1.3: POST Requests with Pydantic (25 min)

Goal: Accept and validate JSON data

Add to `main.py`:

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

@app.post("/items/")
def create_item(item: Item):
    item_dict = item.dict()
    if item.tax:
        price_with_tax = item.price + item.tax
        item_dict["price_with_tax"] = price_with_tax
    return item_dict
```

Exercise 1.3: Test POST Request

Using Swagger UI:

1. Go to `POST /items/`
2. Click "Try it out"
3. Enter JSON:

```
{
  "name": "Laptop",
  "description": "A powerful laptop",
  "price": 999.99,
  "tax": 99.99
}
```

4. Click "Execute"

Using curl:

Part 1 Checkpoint

What you've learned:

- Creating GET and POST endpoints
- Path parameters (`/items/{id}`)
- Query parameters (`?q=value`)
- Request body validation with Pydantic
- Automatic interactive documentation

Next: Build real ML APIs!

Part 2: Data Validation

Building robust APIs with Pydantic

Exercise 2.1: Field Validation (30 min)

Goal: Add constraints to inputs

Create `models.py`:

```
from pydantic import BaseModel, Field, validator
from typing import List

class PredictionInput(BaseModel):
    sepal_length: float = Field(
        ...,  # Required
        gt=0,  # Greater than 0
        lt=10,  # Less than 10
        description="Sepal length in cm"
    )
    sepal_width: float = Field(gt=0, lt=10)
    petal_length: float = Field(gt=0, lt=10)
    petal_width: float = Field(gt=0, lt=10)

    class Config:
        schema_extra = {
            "example": {
                "sepal_length": 5.1,
```

Exercise 2.1: Custom Validators

Add custom validation logic:

```
class PredictionInput(BaseModel):
    sepal_length: float = Field(gt=0, lt=10)
    sepal_width: float = Field(gt=0, lt=10)
    petal_length: float = Field(gt=0, lt=10)
    petal_width: float = Field(gt=0, lt=10)

    @validator('sepal_length')
    def sepal_length_reasonable(cls, v):
        if v < 4.0 or v > 8.0:
            raise ValueError('Sepal length outside typical range')
        return v

    @validator('petal_length')
    def petal_length_reasonable(cls, v):
        if v < 1.0 or v > 7.0:
            raise ValueError('Petal length outside typical range')
        return v
```

Exercise 2.1: Test Validation

Add endpoint in `main.py`:

```
from models import PredictionInput

@app.post("/validate")
def validate_input(input_data: PredictionInput):
    return {
        "status": "valid",
        "data": input_data.dict()
    }
```

Test with invalid data:

```
{
  "sepal_length": -1.0,  // Negative!
  "sepal_width": 3.5,
  "petal_length": 1.4,
  "petal_width": 0.2
```

Part 2 Checkpoint

Validation patterns learned:

- Field constraints (gt, lt, ge, le)
- Required vs optional fields
- Custom validators
- Example schemas for documentation

Part 3: Serving ML Models

The core of ML APIs

Exercise 3.1: Train and Save Model (20 min)

Goal: Create a model to serve

Create `train_model.py`:

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
import joblib

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=42
)

# Train model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate
accuracy = model.score(X_test, y_test)
```

Exercise 3.2: Load Model in API (25 min)

Goal: Serve the trained model

Update `main.py`:

```
import joblib
from pathlib import Path

# Global variable for model
model = None

@app.on_event("startup")
def load_model():
    global model
    model_path = Path("iris_model.pkl")
    if model_path.exists():
        model = joblib.load(model_path)
        print("Model loaded successfully!")
    else:
        print("Warning: Model file not found!")

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

Exercise 3.3: Prediction Endpoint (30 min)

Goal: Complete ML prediction API

Add prediction endpoint:

```
from models import PredictionInput

class PredictionOutput(BaseModel):
    species: str
    confidence: float
    probabilities: List[float]

SPECIES_NAMES = ["setosa", "versicolor", "virginica"]

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

    # Prepare input
    features = [[
        input_data.sepal_length,
        input_data.sepal_width,
        input_data.petal_length,
        input_data.petal_width
    ]]

    # Predict
    prediction = model.predict(features)[0]
```

Exercise 3.3: Test Predictions

Test via Swagger UI with this example:

```
{
  "sepal_length": 5.1,
  "sepal_width": 3.5,
  "petal_length": 1.4,
  "petal_width": 0.2
}
```

Expected output:

```
{
  "species": "setosa",
  "confidence": 1.0,
  "probabilities": [1.0, 0.0, 0.0]
}
```


Try other examples from the iris dataset!

Part 3 Checkpoint

ML API patterns learned:

- Loading models at startup
- Global model variable
- Structured input/output with Pydantic
- Error handling (model not loaded)
- Health check endpoint

Part 4: Error Handling

Building robust production APIs

Exercise 4.1: Handle Errors Gracefully (25 min)

Goal: Catch and report errors properly

Add error handling to prediction:

```python
import logging

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

@app.post("/predict", response_model=PredictionOutput)
def predict(input_data: PredictionInput):
    try:
        if model is None:
            logger.error("Prediction failed: Model not loaded")
            raise HTTPException(
                status_code=503,
                detail="Model not available. Please try again later."
            )

        # Log request
        logger.info(f"Prediction request: {input_data.dict()}")

        # Prepare input
        features = [[
            input_data.sepal_length,
            input_data.sepal_width,
            input_data.petal_length,
            input_data.petal_width
        ]]

        # Predict
        prediction = model.predict(features)[0]
        probabilities = model.predict_proba(features)[0]

        result = {
            "species": SPECIES_NAMES[prediction],
            "confidence": float(max(probabilities)),
            "probabilities": probabilities.tolist()
        }

        logger.info(f"Prediction result: {result['species']}")
```

Exercise 4.2: Add Readiness Check (15 min)

Goal: Verify API is ready to serve requests

Add thorough readiness check:

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

    # Test prediction with known input
    try:
        test_input = [[5.1, 3.5, 1.4, 0.2]]
        _ = model.predict(test_input)
        return {
            "status": "ready",
            "message": "API is ready to serve requests"
        }
    except Exception as e:
```

Part 4 Checkpoint

Error handling patterns:

- Try-except blocks
- Structured logging
- HTTP status codes (503 for unavailable, 500 for errors)
- Health vs readiness checks
- Detailed error messages for debugging

Part 5: Testing

Automated verification of your API

Exercise 5.1: Unit Tests (30 min)

Goal: Write tests for your API

Create `test_api.py`:

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

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello, FastAPI!"}

def test_health_check():
    response = client.get("/health")
    assert response.status_code == 200
    data = response.json()
    assert "status" in data
    assert "model_loaded" in data
```

Exercise 5.1: Test Predictions

Add prediction tests:

```python
def test_predict_setosa():
    """Test prediction for setosa species"""
    payload = {
        "sepal_length": 5.1,
        "sepal_width": 3.5,
        "petal_length": 1.4,
        "petal_width": 0.2
    }
    response = client.post("/predict", json=payload)
    assert response.status_code == 200
    data = response.json()
    assert "species" in data
    assert "confidence" in data
    assert "probabilities" in data
    assert data["species"] == "setosa"

def test_predict_invalid_input():
    """Test that invalid input returns 422"""
    payload = {
        "sepal_length": -1.0,  # Invalid (negative)
        "sepal_width": 3.5,
        "petal_length": 1.4,
        "petal_width": 0.2
    }
    response = client.post("/predict", json=payload)
    assert response.status_code == 422

def test_predict_missing_field():
    """Test that missing required field returns 422"""
    payload = {
        "sepal_length": 5.1,
        # Missing other required fields
```

Exercise 5.1: Run Tests

Execute tests:

```
pytest test_api.py -v
```

Expected output:

```
test_api.py::test_read_root PASSED
test_api.py::test_health_check PASSED
test_api.py::test_readiness_check PASSED
test_api.py::test_predict_setosa PASSED
test_api.py::test_predict_invalid_input PASSED
test_api.py::test_predict_missing_field PASSED
```

Add more tests for different iris species!

Part 5 Checkpoint

Testing skills gained:

- Using TestClient for API tests
- Testing successful requests
- Testing error cases
- Validating response structure
- Running tests with pytest

Part 6: Advanced Features

Production-ready enhancements

Exercise 6.1: Add API Metadata (15 min)

Goal: Improve documentation

Update app initialization in `main.py`:

```
app = FastAPI(
    title="Iris Species Prediction API",
    description="""
    API for predicting Iris flower species based on measurements.

    ## Features
    * Predict species from sepal/petal measurements
    * Input validation with clear error messages
    * Health and readiness checks
    * Comprehensive documentation

    ## Model
    * Algorithm: Random Forest Classifier
    * Accuracy: ~95%
    * Training data: Iris dataset (150 samples)
    """,
    version="1.0.0",
```

Exercise 6.2: Add Batch Prediction (20 min)

Goal: Predict multiple samples at once

Add batch endpoint:

```python
class BatchPredictionInput(BaseModel):
    samples: List[PredictionInput]

class BatchPredictionOutput(BaseModel):
    predictions: List[PredictionOutput]
    count: int

@app.post("/predict/batch", response_model=BatchPredictionOutput)
def predict_batch(input_data: BatchPredictionInput):
    if model is None:
        raise HTTPException(503, detail="Model not loaded")

    results = []
    for sample in input_data.samples:
        features = [[
            sample.sepal_length,
            sample.sepal_width,
            sample.petal_length,
            sample.petal_width
        ]]

        prediction = model.predict(features)[0]
        probabilities = model.predict_proba(features)[0]

        results.append({
            "species": SPECIES_NAMES[prediction],
            "confidence": float(max(probabilities)),
```

Exercise 6.2: Test Batch Prediction

Test with multiple samples:

```
{
  "samples": [
    {
      "sepal_length": 5.1,
      "sepal_width": 3.5,
      "petal_length": 1.4,
      "petal_width": 0.2
    },
    {
      "sepal_length": 7.0,
      "sepal_width": 3.2,
      "petal_length": 4.7,
      "petal_width": 1.4
    }
  ]
}
```

Exercise 6.3: Add CORS Support (10 min)

Goal: Allow web apps to call your API

Add CORS middleware:

```
from fastapi.middleware.cors import CORSMiddleware

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

Now: Any web application can call your API!

Part 6 Checkpoint

Advanced features added:

- Rich API metadata and documentation
- Batch prediction endpoint
- CORS support for web apps
- Production-ready configuration

Part 7: Deployment

Running in production

Exercise 7.1: Create Requirements File (10 min)

Goal: Document dependencies

Create `requirements.txt`:

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
scikit-learn==1.3.2
joblib==1.3.2
pytest==7.4.3
httpx==0.25.1
```

Install all:

```
pip install -r requirements.txt
```

Exercise 7.2: Production Server (15 min)

Goal: Run with production-grade server

Stop development server (Ctrl+C)

Run with Uvicorn (production):

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

Options explained:

- `--host 0.0.0.0` : Accept connections from any IP
- `--port 8000` : Listen on port 8000
- `--workers 4` : Run 4 worker processes

Test: API still works, but faster and more robust!

Exercise 7.3: Docker Container (Optional, 20 min)

Goal: Containerize your API

Create `Dockerfile`:

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

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

Build:

```
docker build -t iris-api .
```

Part 7 Checkpoint

Deployment skills:

- Creating requirements file
- Running with production server (Uvicorn)
- Multi-worker configuration
- Docker containerization (optional)

Lab Wrap-up

What you've accomplished:

✅
Built complete REST API with FastAPI

✅
Implemented input validation with Pydantic

✅
Served ML model via HTTP endpoint

✅
Added error handling and logging

✅
Wrote comprehensive tests

✅
Added batch predictions

✅
Deployed with production server

Deliverables

Submit the following:

1. **Source code**:
    - `main.py` (FastAPI application)
    - `models.py` (Pydantic models)
    - `train_model.py` (Model training script)
    - `test_api.py` (Tests)
    - `requirements.txt`
2. **Artifacts**:
    - `iris_model.pkl` (Trained model)
3. **Documentation**:

Bonus Challenges

If you finish early, try these:

1. **Add authentication**: Require API key for predictions
2. **Add rate limiting**: Limit requests per minute
3. **Add caching**: Cache predictions for repeated inputs
4. **Add monitoring**: Track prediction latency
5. **Support multiple models**: Add version parameter
6. **Add database**: Store all predictions in SQLite
7. **Create frontend**: Simple HTML form to test API

Troubleshooting

Model not loading:

- Ensure `train_model.py` was run first
- Check `iris_model.pkl` exists in same directory

Tests failing:

- Restart the server
- Check model is loaded
- Verify input data format

Port already in use:

```
# Use different port
uvicorn main:app --port 8001
```

Resources

Documentation:

- FastAPI: https://fastapi.tiangolo.com
- Pydantic: https://docs.pydantic.dev
- Uvicorn: https://www.uvicorn.org

Next steps:

- Deploy to cloud (Render, Railway, AWS)
- Add authentication and authorization
- Implement request logging
- Monitor with Prometheus
- Scale with Kubernetes