Lesson 42: REST APIs with FastAPI

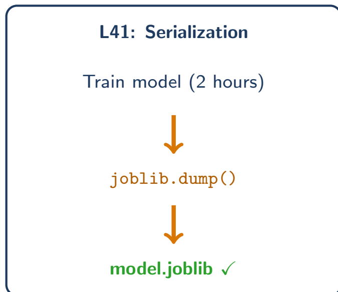
Data Science with Python – BSc Course

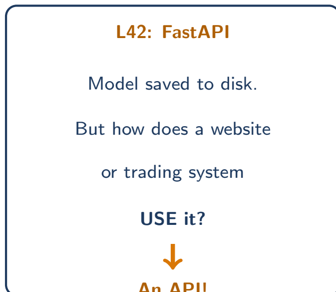
Data Science Program

BSc Course

45 Minutes

Previously: Saving Your Work





L41 saved the model. Now: how does the outside world send it data and get predictions back?

From file on disk to live service – the API bridge

Learning Objectives

The Problem: Your model works in a notebook. But how does it serve 1,000 requests per second from a website, mobile app, or trading system?

After this lesson, you will be able to:

- Create REST API endpoints with FastAPI
- Design input/output schemas with Pydantic validation
- Serve ML model predictions via HTTP requests
- Document APIs automatically with Swagger/OpenAPI

Finance Application: Serving real-time stock predictions to trading systems and dashboards

Your Model Works in a Notebook...

The Problem:

- Your model lives in a Jupyter notebook on YOUR laptop
- A website needs predictions – it can't run your notebook
- A mobile app needs predictions – it doesn't have Python
- A trading system needs predictions – in **milliseconds**

The Solution: An API

- API = Application Programming Interface
- A **contract**: "Send me data in THIS format, I'll respond in THAT format"
- Any programming language can call your API over HTTP
- Your model stays in Python; the world talks to it via HTTP

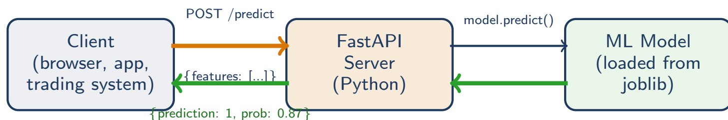
Analogy – A Restaurant Waiter:

- You (client) tell the waiter (API) what you want
- The kitchen (model) prepares the response
- You don't need to know how the kitchen works – just the menu

---

API = universal interface. Any language, any device, any platform can use your model.

REST API: How It Works



HTTP Methods:

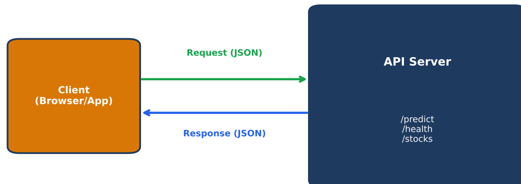
- **GET:** Retrieve information ("Show me model info")
- **POST:** Send data for processing ("Here are features, give me a prediction")

Client sends features via POST, API returns prediction as JSON. Simple.

API Architecture Overview

Components of an ML API

- Request handling: parse incoming JSON data
- Validation: check data types and ranges
- Prediction: run model on validated input



FastAPI handles parsing, validation, and documentation automatically

FastAPI: Hello World

Your First API in 5 Lines:

- `from fastapi import FastAPI`
- `app = FastAPI()`
- `@app.get("/")`
- `def root():`
- `  return {"message": "Hello, World!"}`

Run It:

- `uvicorn main:app --reload`
- Visit `http://localhost:8000` – see JSON response
- Visit `http://localhost:8000/docs` – interactive Swagger UI

Why FastAPI (Not Flask)?

- Type hints → automatic validation
- Auto-generated docs at `/docs`
- Speed comparable to Node.js and Go
- Modern async/await support

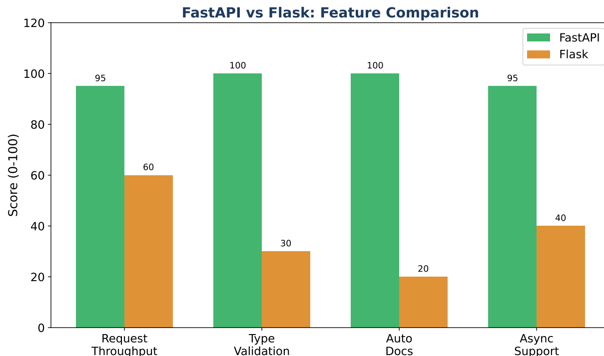
---

5 lines of code gives you a running API with automatic documentation

FastAPI vs Flask

Why FastAPI is the Modern Choice

- Flask: mature, huge ecosystem, simpler for basic apps
- FastAPI: type-safe, faster, auto-docs, async-native
- For ML APIs specifically: FastAPI wins on validation and speed



Flask for general web apps, FastAPI for ML APIs and data services.

Designing Your API Endpoints

Standard ML API Structure:

- **Health check:** `GET /health` – “Is the API running?”
- **Single prediction:** `POST /predict` – one set of features
- **Batch prediction:** `POST /predict/batch` – multiple at once
- **Model info:** `GET /model/info` – version, features, performance

URL Design Best Practices:

- Use nouns, not verbs: `/predictions` not `/getPrediction`
- Version your API: `/v1/predict`, `/v2/predict`
- Return meaningful HTTP status codes:
  - 200: Success
  - 422: Invalid input (Pydantic catches this)
  - 500: Server error
  - 503: Model not loaded

Clean endpoint design makes your API intuitive and maintainable

Path Parameters and Query Parameters

Path Parameters – Part of the URL:

- `@app.get("/stocks/{ticker}")`
- `def get_stock(ticker: str):`
- `  return {"ticker": ticker, "price": lookup(ticker)}`
- Usage: `GET /stocks/AAPL`

Query Parameters – After the ?:

- `@app.get("/stocks/{ticker}/history")`
- `def history(ticker: str, days: int = 30):`
- `  return get_history(ticker, days)`
- Usage: `GET /stocks/AAPL/history?days=90`

Key Difference:

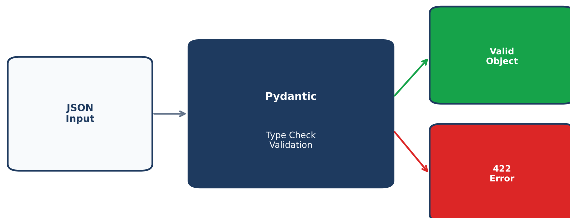
- Path params: required, identify the resource
- Query params: optional, filter/modify the request

Path params identify WHAT. Query params modify HOW.

Pydantic: Automatic Input Validation

Define WHAT Your API Expects

- Python type hints become automatic validators
- Wrong types rejected with clear error messages
- No manual validation code needed



Pydantic Schemas: Input and Output

Input Schema:

- `from pydantic import BaseModel`
- `class PredictionInput(BaseModel):`
- `  volatility: float`
- `  momentum: float`
- `  volume_ratio: float`

Output Schema:

- `class PredictionOutput(BaseModel):`
- `  prediction: int`
- `  probability: float`
- `  model_version: str`

What Happens with Bad Input?

- Send `{"volatility": "high"}` → 422 error: "value is not a valid float"
- Automatic, detailed error messages – no code from you

Define schemas once. Pydantic validates every request automatically.

Checkpoint: GET vs POST

Which HTTP method would you use?

1. Checking if the API is healthy and running?
2. Sending stock features to get a buy/sell prediction?
3. Retrieving the current model version and accuracy?
4. Uploading a CSV file of stocks for batch predictions?

Answers:

1. **GET** /health – just retrieving status, no data sent
2. **POST** /predict – sending feature data for processing
3. **GET** /model/info – retrieving information
4. **POST** /predict/batch – sending data for processing

Rule: GET = read/retrieve. POST = send data for processing.

GET retrieves. POST processes. ML predictions almost always use POST.

Loading Your Model in FastAPI

Load Once at Startup (Not Every Request!)

- `import joblib`
- `from contextlib import asynccontextmanager`
- `model = None`

Lifespan Event (Modern FastAPI):

- `@asynccontextmanager`
- `async def lifespan(app):`
- `  global model`
- `  model = joblib.load("model.joblib")`
- `  yield # app runs here`
- `app = FastAPI(lifespan=lifespan)`

Why at Startup?

- Loading a model: 300ms. Loading per request at 1000 req/s = disaster
- Load once, predict many – same principle as L41

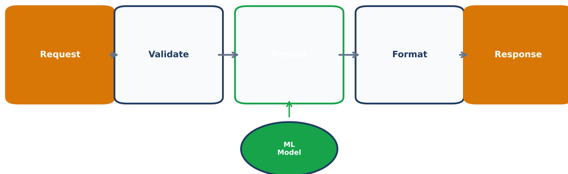
Load model ONCE at startup. Predict on every request. Never reload per-request.

The Prediction Endpoint

Request → Validate → Predict → Respond

- Client sends JSON features via POST
- Pydantic validates, model predicts, JSON returned
- Complete round-trip in single-digit milliseconds

ML Prediction Endpoint Flow



The predict endpoint is the core of any ML API – simple but powerful

Prediction Endpoint: Code

Complete Implementation:

- `@app.post("/predict", response_model=PredictionOutput)`
- `def predict(input:  PredictionInput):`
- `  features = [[input.volatility,`
- `    input.momentum, input.volume_ratio]]`
- `  pred = model.predict(features)[0]`
- `  prob = model.predict_proba(features)[0].max()`
- `  return PredictionOutput(`
- `    prediction=int(pred),`
- `    probability=float(prob),`
- `    model_version="v1")`

What FastAPI Does Automatically:

- Parses JSON body into PredictionInput object
- Validates all fields (type, required)
- Serializes PredictionOutput to JSON response

FastAPI: parse, validate, serialize – all automatic from your type hints

Async: Handling Many Requests

The Problem:

- 100 users send predictions at the same time
- Synchronous: serve one at a time (others wait)
- Asynchronous: handle many concurrently

FastAPI Supports Both:

- `def predict(...)` – synchronous (fine for CPU-bound ML)
- `async def predict(...)` – async (great for I/O-bound tasks)

When to Use Async:

- Database queries, external API calls, file I/O
- NOT for CPU-heavy model inference (use sync + multiple workers)

For ML APIs: Use `def` (sync) with multiple uvicorn workers:

- `uvicorn main:app --workers 4` – 4 parallel processes

ML prediction is CPU-bound: use sync endpoints with multiple workers

Error Handling: Graceful Failure

Three Layers of Protection:

1. **Pydantic:** Wrong types → 422 (automatic)
2. **Custom checks:** Business logic → 400
3. **Try/except:** Model errors → 500

Custom Error Handling:

- `from fastapi import HTTPException`
- `if model is None:`
- `  raise HTTPException(503, detail="Model not loaded")`
- `try:`
- `  pred = model.predict(features)`
- `except Exception as e:`
- `  raise HTTPException(500, detail=str(e))`

Finance: Log every error with timestamp for regulatory audit trail

---

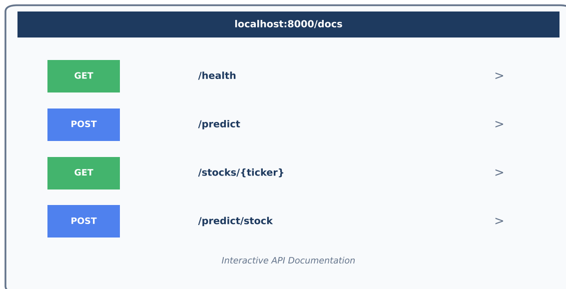
Never return raw tracebacks. Always return structured JSON errors.

Swagger: Auto-Generated API Docs

Documentation You Don't Have to Write

- Visit `/docs` for interactive Swagger UI
- "Try it out" button to test endpoints live
- Generated from your Pydantic schemas automatically

Swagger UI: Auto-Generated API Docs

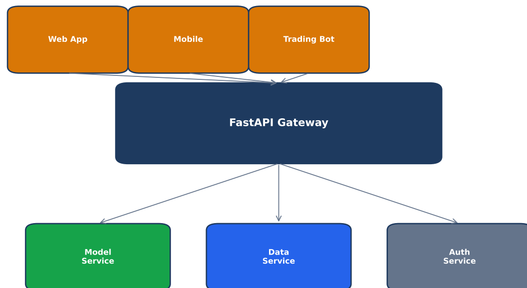


Swagger at /docs, ReDoc at /redoc. Both generated automatically from your code.

Finance: Real-Time Prediction API

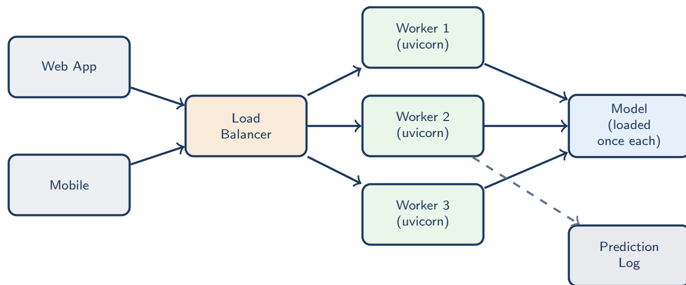
Complete Stock Prediction Service

- Input: RSI, MACD, volatility, volume_ratio, momentum
- Output: buy/sell/hold + confidence score + model version
- Logging: every prediction timestamped for audit



A complete ML API in under 50 lines of Python – FastAPI makes it simple

Production API Architecture



Scale: Multiple uvicorn workers behind a load balancer. Each worker loads the model once at startup.

Production: load balancer + multiple workers + shared model + prediction logging

Hands-On Exercise (25 min)

Task: Build a Stock Prediction API

1. Load your saved model from L41 at startup
2. Create FastAPI app with `GET /health` and `POST /predict`
3. Define Pydantic schemas for input features and output prediction
4. Add error handling for missing model and invalid inputs
5. Test via Swagger docs at `http://localhost:8000/docs`

Test with curl:

- `curl -X POST http://localhost:8000/predict`
- `-H "Content-Type: application/json"`
- `-d '{"volatility": 0.02, "momentum": 0.5, "volume_ratio": 1.2}'`

Deliverable: Running API + screenshot of Swagger docs with test prediction.

Extension: Add a /predict/batch endpoint for multiple predictions at once

Your Model Goes Online

Before L42	**After L42**
"My model is great!"	"My model is great!"
colleague: "Can I use it?"	colleague: "Can I use it?"
"Sure, install Python,"	**"POST to /predict"**
"sklearn, numpy. . ."	"here's the Swagger docs"
colleague left the chat	works from any language!

Your model is now a service. But only developers can use an API. Next: a dashboard for humans.

APIs are for machines. Next lesson: Streamlit dashboards for humans.

Lesson Summary

Problem Solved: We can now serve ML predictions via a web API that any application can consume.

Key Takeaways:

- **FastAPI:** modern, fast, auto-documented Python API framework
- **Pydantic:** schemas validate input/output automatically
- **Startup loading:** load model once, predict on every request
- **Swagger:** interactive docs generated at `/docs`
- **Error handling:** structured JSON errors, never raw tracebacks

Next Lesson: Streamlit Dashboards (L43) – interactive web apps that humans can see and use

---

L41: Save. L42: Serve via API. L43: Build a dashboard. L44: Deploy to cloud.