**Comprehensive FastAPI Guide for Data Engineering**

**Introduction to FastAPI & RESTful Services**

FastAPI is a modern, high-performance Python web framework designed for building RESTful APIs. It leverages standard Python type hints to provide automatic data validation, interactive documentation, and excellent developer experience. For data engineering, FastAPI excels at creating robust endpoints for ETL pipelines, real-time data ingestion, and model serving, all with minimal boilerplate and maximum performance.

**1. What Is FastAPI and Why Use It for Data Engineering**

FastAPI is built on top of Starlette for web handling and Pydantic for data validation. It supports synchronous and asynchronous programming paradigms using Python’s async/await, enabling non-blocking I/O operations that suit high-throughput data tasks. Key benefits for data engineering include:

* **Automatic validation**: Request and response payloads are validated against Pydantic models, reducing runtime errors.
* **Interactive docs**: Out-of-the-box Swagger UI and ReDoc let you explore and test APIs without extra setup.
* **High throughput**: Benchmarks show FastAPI rivals Node.js and Go in request handling, making it ideal for streaming and batch data services.
* **Minimal code**: Declarative route definitions and dependency injection minimize boilerplate, accelerating development.

**2. Features**

**2.1 Asynchronous Support**

FastAPI endpoints can be defined as async def, allowing the server to handle other requests during long-running I/O (e.g., database queries, file uploads).

@app.get("/data/")  
async def fetch\_data():  
 result = await database.fetch\_all("SELECT \* FROM table")  
 return result

**2.2 Type Hints**

Function signatures annotated with Python’s typing module enable Pydantic to parse and validate incoming data.

from typing import List  
  
@app.post("/numbers/")  
async def sum\_numbers(nums: List[int]):  
 return {"sum": sum(nums)}

**2.3 Speed**

Underpinned by Uvicorn (ASGI server) and Starlette, FastAPI handles thousands of requests per second with low latency, critical for real-time dashboards and streaming pipelines.

**3. Setting Up FastAPI Environment**

**3.1 Python Virtual Environment**

Create a dedicated virtual environment to isolate dependencies:

python3 -m venv venv  
source venv/bin/activate

**3.2 Install Dependencies**

Install FastAPI and Uvicorn:

pip install fastapi uvicorn

Optionally add database drivers, e.g., pymysql for MySQL or asyncpg for PostgreSQL.

**3.3 Project Structure**

Adopt a modular layout for scalability:

project/  
├── app/  
│ ├── main.py # Application entry point  
│ ├── models.py # Pydantic and SQLAlchemy models  
│ ├── routers/ # Route modules  
│ │ └── etl.py  
│ ├── services/ # Business logic and ETL functions  
│ │ └── etl\_job.py  
│ ├── db.py # Database connection and session  
│ └── utils.py # Logging, helpers  
└── requirements.txt

List fixed versions in requirements.txt for reproducibility.

**4. Creating Your First FastAPI App**

In app/main.py, instantiate the application:

from fastapi import FastAPI  
  
app = FastAPI(  
 title="Data Engineering API",  
 description="ETL and data services built with FastAPI",  
 version="1.0.0"  
)  
  
@app.get("/")  
async def root():  
 return {"message": "Welcome to the Data Engineering API"}

Run in development mode with auto-reload:

uvicorn app.main:app --reload

**5. Basic GET and POST Endpoints**

**5.1 GET Endpoint with Path & Query Parameters**

from fastapi import FastAPI  
from typing import Optional  
  
app = FastAPI()  
  
@app.get("/items/{item\_id}")  
async def read\_item(item\_id: int, q: Optional[str] = None):  
 return {"item\_id": item\_id, "query": q}

* **Path parameter**: item\_id extracted from URL.
* **Query parameter**: q optional search term.

**5.2 POST Endpoint with Pydantic Model**

Create a Pydantic schema in models.py:

from pydantic import BaseModel  
  
class Item(BaseModel):  
 name: str  
 price: float  
 tags: list[str] = []

Use it in main.py:

from fastapi import FastAPI  
from .models import Item  
  
app = FastAPI()  
  
@app.post("/items/")  
async def create\_item(item: Item):  
 return {"item": item}

This automatically enforces types and returns JSON matching the model.

**6. API Development with FastAPI**

**6.1 Data Models with Pydantic**

Define input/output schemas:

from pydantic import BaseModel, EmailStr  
  
class UserBase(BaseModel):  
 email: EmailStr  
 is\_active: bool = True  
  
class UserCreate(UserBase):  
 password: str  
  
class UserOut(UserBase):  
 id: int  
  
 class Config:  
 orm\_mode = True

* **orm\_mode**: allows SQLAlchemy model integration.

**6.2 Form and File Uploads**

from fastapi import File, UploadFile, Form  
from fastapi import FastAPI  
  
app = FastAPI()  
  
@app.post("/upload/")  
async def upload\_file(  
 file: UploadFile = File(...),  
 description: str = Form(...)  
):  
 content = await file.read()  
 return {"filename": file.filename, "size": len(content), "desc": description}

* **UploadFile** streams large files efficiently.

**6.3 Error Handling and Validation**

Raise standardized HTTP errors:

from fastapi import HTTPException, status  
  
@app.get("/secure/")  
async def secure\_endpoint(token: str):  
 if token != "expected":  
 raise HTTPException(  
 status\_code=status.HTTP\_401\_UNAUTHORIZED,  
 detail="Invalid authentication"  
 )  
 return {"access": "granted"}

**6.4 Middleware and Dependencies**

**Middleware Example**

import time  
from fastapi import Request, FastAPI  
  
app = FastAPI()  
  
@app.middleware("http")  
async def add\_process\_time(request: Request, call\_next):  
 start = time.time()  
 response = await call\_next(request)  
 response.headers["X-Process-Time"] = str(time.time() - start)  
 return response

**Dependency Injection**

from fastapi import Depends  
  
def common\_parameters(q: str | None = None, limit: int = 10):  
 return {"q": q, "limit": limit}  
  
@app.get("/search/")  
async def search(params: dict = Depends(common\_parameters)):  
 return params

* **Dependencies**: reusable parameter definitions, authentication, database sessions.

**6.5 Automatic Documentation**

* **Swagger UI**: navigate to /docs.
* **ReDoc**: navigate to /redoc.

Interactive docs reflect path operations, schemas, and examples derived from type hints.

**7. Integration with Data Engineering Workflows**

**7.1 Integrating FastAPI with ETL Jobs**

In services/etl\_job.py:

import pandas as pd  
  
def etl\_job(source\_path: str, target\_path: str) -> int:  
 df = pd.read\_csv(source\_path)  
 # Example transformation  
 df["processed"] = True  
 df.to\_parquet(target\_path)  
 return len(df)

Expose via API:

from fastapi import APIRouter  
from .services.etl\_job import etl\_job  
from pydantic import BaseModel  
  
router = APIRouter(prefix="/etl")  
  
class ETLParams(BaseModel):  
 source: str  
 target: str  
  
@router.post("/")  
async def run\_etl(params: ETLParams):  
 rows = etl\_job(params.source, params.target)  
 return {"status": "completed", "rows\_processed": rows}

Include router in main.py:

from fastapi import FastAPI  
from .routers import etl  
  
app = FastAPI()  
app.include\_router(etl.router)

**7.2 Working with Pandas and NumPy Inside Endpoints**

from fastapi import FastAPI  
from pydantic import BaseModel  
import pandas as pd  
import numpy as np  
  
app = FastAPI()  
  
class StatsRequest(BaseModel):  
 data: list[float]  
  
@app.post("/stats/")  
async def compute\_stats(req: StatsRequest):  
 arr = np.array(req.data)  
 df = pd.DataFrame(arr, columns=["values"])  
 return {  
 "count": int(df["values"].count()),  
 "mean": df["values"].mean(),  
 "std\_dev": df["values"].std(),  
 "min": df["values"].min(),  
 "max": df["values"].max()  
 }

**7.3 Logging and Monitoring APIs**

Configure standard logging:

import logging  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger("data\_api")  
  
@app.get("/log-demo/")  
async def log\_demo():  
 logger.info("Log demo endpoint accessed")  
 return {"message": "Check your console or log file"}

For production, integrate with tools like **Prometheus**, **Grafana**, or **Sentry** for metrics and error tracking.

**8. Database Integration**

**8.1 Connecting FastAPI with MySQL**

Install dependencies:

pip install sqlalchemy pymysql

In db.py:

from sqlalchemy import create\_engine  
from sqlalchemy.orm import sessionmaker  
  
SQLALCHEMY\_DATABASE\_URL = "mysql+pymysql://user:password@localhost:3306/dbname"  
  
engine = create\_engine(SQLALCHEMY\_DATABASE\_URL, pool\_pre\_ping=True)  
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

**8.2 Defining SQLAlchemy Models**

In models.py:

from sqlalchemy import Column, Integer, String  
from sqlalchemy.ext.declarative import declarative\_base  
  
Base = declarative\_base()  
  
class User(Base):  
 \_\_tablename\_\_ = "users"  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String(100), nullable=False)  
 email = Column(String(255), unique=True, index=True, nullable=False)

Create tables:

from .db import engine  
from .models import Base  
  
Base.metadata.create\_all(bind=engine)

**8.3 Dependency for Database Session**

from fastapi import Depends  
from sqlalchemy.orm import Session  
from .db import SessionLocal  
  
def get\_db():  
 db = SessionLocal()  
 try:  
 yield db  
 finally:  
 db.close()

Inject db: Session = Depends(get\_db) into path operations.

**9. Building CRUD APIs for Data Engineering Use Cases**

**9.1 Create Operation**

from fastapi import APIRouter, Depends, HTTPException, status  
from sqlalchemy.orm import Session  
from .models import User  
from .schemas import UserCreate, UserOut  
from .db import get\_db  
  
router = APIRouter(prefix="/users", tags=["users"])  
  
@router.post("/", response\_model=UserOut, status\_code=status.HTTP\_201\_CREATED)  
def create\_user(user: UserCreate, db: Session = Depends(get\_db)):  
 db\_user = User(name=user.name, email=user.email)  
 db.add(db\_user)  
 db.commit()  
 db.refresh(db\_user)  
 return db\_user

**9.2 Read Operation**

@router.get("/{user\_id}", response\_model=UserOut)  
def read\_user(user\_id: int, db: Session = Depends(get\_db)):  
 db\_user = db.query(User).filter(User.id == user\_id).first()  
 if not db\_user:  
 raise HTTPException(status\_code=404, detail="User not found")  
 return db\_user

**9.3 Update Operation**

from .schemas import UserUpdate  
  
@router.put("/{user\_id}", response\_model=UserOut)  
def update\_user(user\_id: int, upd: UserUpdate, db: Session = Depends(get\_db)):  
 db\_user = db.query(User).get(user\_id)  
 if not db\_user:  
 raise HTTPException(status\_code=404, detail="User not found")  
 for key, val in upd.dict(exclude\_unset=True).items():  
 setattr(db\_user, key, val)  
 db.commit()  
 db.refresh(db\_user)  
 return db\_user

**9.4 Delete Operation**

@router.delete("/{user\_id}", status\_code=status.HTTP\_204\_NO\_CONTENT)  
def delete\_user(user\_id: int, db: Session = Depends(get\_db)):  
 db\_user = db.query(User).get(user\_id)  
 if not db\_user:  
 raise HTTPException(status\_code=404, detail="User not found")  
 db.delete(db\_user)  
 db.commit()

Include router in main.py:

app.include\_router(router)