Highlighting FastAPI: High-Performance Data Ingestion API

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints. In your data platform, it serves as the crucial **ingestion layer**, providing robust and well-documented endpoints for receiving disparate data (e.g., financial transactions, insurance claims) from external sources before it enters your streaming pipeline.
This guide will demonstrate basic and advanced use cases of FastAPI, leveraging your **Advanced Track** local environment setup and its integration with Kafka and observability tools.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**.

Basic Use Case: Receiving and Acknowledging Data Ingestion

Objective: To demonstrate FastAPI's core function: providing a well-defined HTTP endpoint to receive data and return a success acknowledgment.
Role in Platform: Act as the secure, high-performance gateway for all incoming data, translating external requests into internal data platform events.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root.
2. **Verify FastAPI is accessible:** Check Docker logs for the fastapi_ingestor container (docker compose logs fastapi_ingestor). Its health check endpoint should be reachable at http://localhost:8000/health.
3. **Review FastAPI application code:** Your fastapi_app/app/main.py should define the API endpoints and Pydantic models for request bodies.
   Example fastapi_app/app/main.py (conceptual, core parts):
   # fastapi_app/app/main.py
   from fastapi import FastAPI, HTTPException, status
   from pydantic import BaseModel, Field
   from typing import Optional
   from datetime import datetime
   import os
   import json
   from kafka import KafkaProducer # Assumed to be installed in FastAPI container

```
# --- Pydantic Models for Data Contracts ---
class FinancialTransaction(BaseModel):
    transaction_id: str = Field(..., example="FT-20231026-001")
    timestamp: datetime = Field(..., example="2023-10-26T14:30:00Z")
    account_id: str = Field(..., example="ACC-001")
    amount: float = Field(..., gt=0, example=150.75) # Amount must be greater than 0
    currency: str = Field(..., max_length=3, example="USD")
    transaction_type: str = Field(..., example="debit")
    merchant_id: Optional[str] = Field(None, example="MER-XYZ")
    category: Optional[str] = Field(None, example="groceries")

class InsuranceClaim(BaseModel):
    claim_id: str = Field(..., example="IC-20231026-001")
    timestamp: datetime = Field(..., example="2023-10-26T15:00:00Z")
    policy_number: str = Field(..., example="POL-987654")
    claim_amount: float = Field(..., gt=0, example=1000.00)
    claim_type: str = Field(..., example="auto")
    claim_status: str = Field(..., example="submitted")
    customer_id: str = Field(..., example="CUST-ABC")
    incident_date: datetime = Field(..., example="2023-09-15T08:00:00Z")

# --- FastAPI Application Initialization ---
app = FastAPI(
    title="Financial/Insurance Data Ingestor API",
    description="API for ingesting various financial and insurance data into the data
platform.",
    version="1.0.0",
)

# --- Kafka Producer Setup ---
KAFKA_BROKER = os.getenv("KAFKA_BROKER", "kafka:29092") # Default to service
name for Docker Compose
KAFKA_TOPIC_FINANCIAL = os.getenv("KAFKA_TOPIC_FINANCIAL",
"raw_financial_transactions")
KAFKA_TOPIC_INSURANCE = os.getenv("KAFKA_TOPIC_INSURANCE",
"raw_insurance_claims")

# Initialize Kafka producer globally (or use a dependency injection)
try:
    producer = KafkaProducer(
        bootstrap_servers=[KAFKA_BROKER],
        value_serializer=lambda v: json.dumps(v).encode('utf-8'),
        retries=5, # Number of retries on failed sends
```

```
        linger_ms=100, # Batch messages for 100ms
        batch_size=16384 # Batch size in bytes
    )
    print(f"Kafka Producer initialized for broker: {KAFKA_BROKER}")
except Exception as e:
    print(f"Error initializing Kafka Producer: {e}")
    producer = None # Handle case where producer fails to initialize

# --- API Endpoints ---
@app.get("/health", tags=["Monitoring"])
async def health_check():
    """Health check endpoint."""
    return {"status": "healthy", "message": "Welcome to Financial/Insurance Data Ingestor API!"}

@app.post("/ingest-financial-transaction/", status_code=status.HTTP_200_OK, tags=["Ingestion"])
async def ingest_financial_transaction(transaction: FinancialTransaction):
    """
    Ingests a financial transaction record.
    This endpoint uses Pydantic for automatic request validation.
    """
    try:
        if producer:
            producer.send(KAFKA_TOPIC_FINANCIAL, transaction.dict()).get(timeout=10) # Send synchronously for basic verification
            print(f"Financial transaction ingested and sent to Kafka topic '{KAFKA_TOPIC_FINANCIAL}': {transaction.transaction_id}")
        else:
            print("Kafka producer not available. Skipping send.")
        return {"message": "Financial transaction ingested successfully", "transaction_id": transaction.transaction_id}
    except Exception as e:
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail=f"Failed to ingest transaction: {e}")

@app.post("/ingest-insurance-claim/", status_code=status.HTTP_200_OK, tags=["Ingestion"])
async def ingest_insurance_claim(claim: InsuranceClaim):
    """
    Ingests an insurance claim record.
    This endpoint uses Pydantic for automatic request validation.
    """
```

```
    try:
        if producer:
            producer.send(KAFKA_TOPIC_INSURANCE, claim.dict()).get(timeout=10) # Send
synchronously for basic verification
            print(f"Insurance claim ingested and sent to Kafka topic
'{KAFKA_TOPIC_INSURANCE}': {claim.claim_id}")
        else:
            print("Kafka producer not available. Skipping send.")
        return {"message": "Insurance claim ingested successfully", "claim_id":
claim.claim_id}
    except Exception as e:
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail=f"Failed to ingest claim: {e}")
```

Steps to Exercise:

1. Send a valid financial transaction:
   Open a terminal and use curl to send a POST request.
   ```
   curl -X POST -H "Content-Type: application/json" \
     -d '{
         "transaction_id": "FT-001",
         "timestamp": "2024-06-14T10:00:00Z",
         "account_id": "ACC-FIN-001",
         "amount": 100.50,
         "currency": "USD",
         "transaction_type": "purchase"
       }' \
     http://localhost:8000/ingest-financial-transaction/
   ```

2. **Send a valid insurance claim:**
   ```
   curl -X POST -H "Content-Type: application/json" \
     -d '{
         "claim_id": "IC-001",
         "timestamp": "2024-06-14T11:00:00Z",
         "policy_number": "POL-12345",
         "claim_amount": 5000.00,
         "claim_type": "auto",
         "claim_status": "submitted",
         "customer_id": "CUST-001",
         "incident_date": "2024-05-01T09:00:00Z"
       }' \
     http://localhost:8000/ingest-insurance-claim/
   ```

3. **Observe FastAPI logs:** In a separate terminal, watch the logs of the fastapi_ingestor

container:
docker compose logs -f fastapi_ingestor

Verification:

- **HTTP Response:** Both curl commands should return HTTP 200 OK with a JSON response confirming successful ingestion (e.g., {"message": "Financial transaction ingested successfully", "transaction_id": "FT-001"}).
- **FastAPI Logs:** The fastapi_ingestor logs will show messages like "Financial transaction ingested and sent to Kafka topic 'raw_financial_transactions': FT-001", confirming receipt and forwarding.

Advanced Use Case 1: Robust Input Validation and Error Handling

Objective: To demonstrate FastAPI's automatic data validation using Pydantic models, and its ability to return clear, structured error responses for invalid input, preventing bad data from entering the pipeline.
Role in Platform: Act as the first line of defense for data quality, ensuring that incoming data adheres to defined schemas and business rules before further processing.
Setup/Configuration:

1. **Basic Use Case completed:** Ensure FastAPI is running and you have the FinancialTransaction and InsuranceClaim Pydantic models defined as shown above.

Steps to Exercise:

1. **Send an invalid financial transaction (missing required field):**
   curl -X POST -H "Content-Type: application/json" \
   -d '{
   "timestamp": "2024-06-14T10:00:00Z",
   "account_id": "ACC-FIN-001",
   "amount": 100.50,
   "currency": "USD",
   "transaction_type": "purchase"
   }' \
   http://localhost:8000/ingest-financial-transaction/

   (transaction_id is missing)
2. **Send an invalid financial transaction (wrong data type):**
   curl -X POST -H "Content-Type: application/json" \
   -d '{
   "transaction_id": "FT-002",
   "timestamp": "2024-06-14T10:00:00Z",
   "account_id": "ACC-FIN-001",
   "amount": "not_a_number",
   "currency": "USD",

```
      "transaction_type": "purchase"
    }' \
  http://localhost:8000/ingest-financial-transaction/
```

(amount is a string, but expects float)

3. **Send an invalid insurance claim (invalid claim_amount - less than or equal to 0):**
```
curl -X POST -H "Content-Type: application/json" \
  -d '{
      "claim_id": "IC-002",
      "timestamp": "2024-06-14T11:00:00Z",
      "policy_number": "POL-12346",
      "claim_amount": -50.00,
      "claim_type": "health",
      "claim_status": "submitted",
      "customer_id": "CUST-002",
      "incident_date": "2024-05-01T09:00:00Z"
    }' \
  http://localhost:8000/ingest-insurance-claim/
```

(claim_amount violates gt=0 constraint)

Verification:

- **HTTP Response:** All curl commands should return HTTP 422 Unprocessable Entity.
- **Response Body:** The response body will contain a structured JSON error message detailing the validation failures (e.g., "field required", "value is not a valid float", "ensure this value is greater than 0"). This provides clear feedback to API clients.
- **FastAPI Logs:** The fastapi_ingestor logs will show INFO:uvicorn.access:XXX "POST /ingest-... 422 Unprocessable Entity" entries, confirming the validation errors.

Advanced Use Case 2: Asynchronous Kafka Publishing and Idempotency

Objective: To demonstrate how FastAPI can asynchronously publish messages to Kafka and how the ingestion endpoint can conceptually handle idempotency to prevent duplicate processing from client retries.

Role in Platform: Maximize ingestion throughput by not waiting for Kafka acknowledgment (for high-volume fire-and-forget scenarios) and ensure data integrity by gracefully handling re-submission of the same data.

Setup/Configuration:

1. **FastAPI with Kafka Producer:** Ensure fastapi_app/app/main.py has the KafkaProducer setup.
2. **Modify producer.send():** For asynchronous behavior, remove .get(timeout=...) to make it non-blocking. For idempotency, we'll conceptualize using a unique ID.
   Example fastapi_app/app/main.py (modification to ingest_financial_transaction):

```
    # ... inside ingest_financial_transaction endpoint ...
        if producer:
            # Send asynchronously and add a callback for logging success/failure
            # This makes the API endpoint respond faster
            future = producer.send(KAFKA_TOPIC_FINANCIAL, transaction.dict())
            future.add_callback(lambda record_metadata: print(
                f"Successfully sent transaction {transaction.transaction_id} to topic "
                f"{record_metadata.topic} partition {record_metadata.partition} "
                f"offset {record_metadata.offset}"
            ))
            future.add_errback(lambda exc: print(
                f"Failed to send transaction {transaction.transaction_id}: {exc}"
            ))
            print(f"Financial transaction ingestion request acknowledged for
    {transaction.transaction_id}. Publishing to Kafka asynchronously.")
        else:
            print("Kafka producer not available. Skipping send.")
        # Conceptual Idempotency check:
        # In a real system, you'd store transaction_id in a fast lookup (e.g., Redis)
        # and reject if already seen within a short window.
        # For this demo, we'll just acknowledge the request as if it's new.
        return {"message": "Financial transaction accepted for processing",
    "transaction_id": transaction.transaction_id}
    # ... similar for insurance claims ...
```

Steps to Exercise:

1. **Restart FastAPI container:** docker compose restart fastapi_ingestor to apply code changes.
2. **Generate data with simulate_data.py:** Let it run for a while.
    - Observe the fastapi_ingestor logs. You'll see Acknowledging... Publishing to Kafka asynchronously messages *immediately*, followed by Successfully sent... callbacks a moment later. This shows the non-blocking send.
3. **Simulate a retry/duplicate submission:**
    - Pick a transaction_id from a previously successful request (e.g., FT-001).
    - Send the *exact same* request again using curl.
    - **Expected Response:** Even though it's a duplicate, the API will still return HTTP 200 OK and Financial transaction accepted for processing.
    - **Conceptual Idempotency Handling:** While the API endpoint itself accepts it, the downstream Spark job (or Lambda) consuming from Kafka *must* be designed to handle idempotency. This is typically done by using the transaction_id as a primary key for upserts into Delta Lake, ensuring that re-processing the same message doesn't create duplicate records but merely updates the existing one.

Verification:

- **FastAPI Response Time:** The FastAPI endpoint will return responses faster compared to blocking on producer.send().get().
- **Kafka Logs:** The Kafka consumer (Spark job or console consumer) will still receive the duplicate message, but the data platform's later stages (e.g., Delta Lake MERGE INTO operation based on transaction_id) are responsible for ensuring logical idempotency, resulting in only one unique record for FT-001 in the curated zone.

Advanced Use Case 3: API Documentation (Swagger UI) & Observability Integration

Objective: To demonstrate FastAPI's automatic generation of interactive API documentation (Swagger UI/OpenAPI) and its integration with OpenTelemetry/Prometheus for detailed API observability metrics.
Role in Platform: Provide self-service documentation for API consumers (developers, data scientists) and ensure comprehensive monitoring of the ingestion layer for operational teams.
Setup/Configuration:

1. **Ensure fastapi_app/app/main.py is configured with title, description, version for FastAPI (as shown in Basic Use Case).**
2. **Install Prometheus Instrumentator:** Ensure prometheus_fastapi_instrumentator is in fastapi_app/requirements.txt and imported/instrumented in main.py (as hinted in "Highlighting cAdvisor" document's Advanced Use Case 2).
   ```
   # fastapi_app/app/main.py (additions for observability)
   from prometheus_fastapi_instrumentator import Instrumentator
   # ...
   app = FastAPI(
      title="Financial/Insurance Data Ingestor API",
      description="API for ingesting various financial and insurance data into the data platform.",
      version="1.0.0",
   )
   Instrumentator().instrument(app).expose(app) # Exposes metrics on /metrics endpoint by default
   ```
3. **Ensure Grafana Alloy is configured to scrape FastAPI metrics:** Check observability/alloy-config.river for a scrape configuration for fastapi_ingestor on port 8000 at path /metrics.
4. **Ensure Grafana is running and accessible.**

Steps to Exercise:

1. **Access Interactive API Documentation (Swagger UI):**
   - Open your web browser and navigate to http://localhost:8000/docs.
   - **Explore:** You will see a comprehensive, interactive documentation of your FastAPI endpoints, including request schemas, example values, and response codes, generated automatically from your Pydantic models and endpoint decorators. You

can even try out API calls directly from this UI.

2. **Access OpenAPI Specification:**
    - Navigate to http://localhost:8000/openapi.json.
    - **Explore:** This provides the raw OpenAPI (Swagger) specification in JSON format, which can be used by various tools for client code generation, API testing, or API management platforms.
3. **Generate API Traffic:**
    - Run python3 simulate_data.py to continuously send requests to the FastAPI endpoints.
4. **Observe API Metrics in Grafana:**
    - Go to http://localhost:3000.
    - Navigate to your "Health Dashboard" or create a new panel.
    - **Query (PromQL):**
        - http_requests_total{job="fastapi_ingestor"}: To see total requests.
        - http_request_duration_seconds_bucket{job="fastapi_ingestor", le="0.5"}: To see requests completing within 500ms (for latency).
        - rate(http_requests_total{job="fastapi_ingestor", method="POST"}[1m]): To see Requests Per Second for POST methods.
    - **Observe:** The graphs will show real-time metrics for your FastAPI ingestion API, demonstrating its throughput, latency, and overall health.

Verification:

- **Swagger UI:** The interactive documentation is fully functional and reflects your API's endpoints and schemas.
- **OpenAPI Spec:** The raw OpenAPI JSON is accessible and well-formed.
- **Grafana:** Metrics from FastAPI are flowing correctly into Grafana, providing granular insights into the API's performance, allowing operators to monitor its health and identify issues quickly.

This concludes the guide for FastAPI.