Backend Assessment Documentation

Task Overview

The task is to create a dockerized service, **claim_process**, to process insurance claims. This includes transforming a JSON payload representing a single claim input with multiple lines into a relational database (RDB), generating unique IDs for claims, computing the net fee for each claim, and integrating with a downstream service, **payments**, that consumes the computed net fees. Additionally, the solution must include an endpoint to return the top 10 provider NPIs by net fees generated, with rate limiting applied.

Requirements

1. **Transform and Store JSON Payload**:
   - The service should transform a JSON payload representing a single claim input with multiple lines and store it into a relational database (PostgreSQL).
   - An example input in CSV format (`claim_1234.csv`) is provided, with inconsistent capitalization in column names.

2. **Generate Unique ID per Claim**:
   - Each claim must have a unique ID generated.

3. **Compute Net Fee**:
   - Compute the net fee for each claim using the formula: `net fee = provider fees + member coinsurance + member copay - allowed fees`.

4. **Downstream Service (Payments)**:
   - The downstream service will consume the net fee computed by **claim_process**.

5. **Top 10 Providers by Net Fees Endpoint**:
   - Implement an endpoint that returns the top 10 provider NPIs by net fees generated.
   - Apply a rate limiter to this API to allow a maximum of 10 requests per minute.

6. **Data Validation**:
   - Validate that the "submitted procedure" always begins with the letter ‘D’.
   - Validate that the "Provider NPI" is always a 10-digit number.
   - All fields except “quadrant” are required.

Completion of Requirements

1. Transform JSON Payload and Store in RDB:
   - Achieved: Transformed CSV input to JSON and stored it in PostgreSQL database.

2. Generate a Unique ID per Claim:
   - Achieved: Used UUID to generate unique IDs for each claim.
   - Trade-offs: Slightly larger storage requirement for UUIDs.

3. Compute the Net Fee:
   - Achieved: Computed net fee using the formula provided.

4. Downstream Service (Payments) Consumes Net Fee:
   - Achieved: Published net fee to RabbitMQ for consumption by the payments service.
   - Benefits: Decouples claim processing and payment services, allowing independent scaling.
   - Trade-offs: Added complexity of message queue setup and management.
   - Bottlenecks: Potential message queue latency and reliability issues.

5. Endpoint to Return Top 10 Provider NPIs by Net Fees:
   - Achieved
   - By using a priority queue (implemented as a max-heap) to maintain the top 10 providers by net fees, we can ensure that only 10 elements are stored at any given time. The retrieval time for the largest net fees element in the heap is O(1), and the time complexity for maintaining the heap is O(nlogk), where k is 10 and n is the number of elements processed. Here's a detailed breakdown: Insertion and Heap Maintenance: Each insertion operation into the priority queue (min-heap) takes O(logk) time. Heap Size Management: If the heap exceeds 10 elements, we remove the smallest element, which also takes O(logk) time. Overall Time Complexity: Since we perform an insertion and possibly a deletion for each of the *n* n elements, the overall time complexity is O(nlogk). Thus, the priority queue ensures efficient storage and retrieval of the top 10 providers by net fees, with the retrieval of the smallest element in O(1) time and a total time complexity of O(nlogk).

6. **Dockerization**
- Achieved: The services are containerized using Docker, making them easy to deploy and manage.

7. **FastAPI Framework**
- Achieved: Used FastAPI to build the API endpoints.

8. **Rate Limiter**
- Achieved: Implemented rate limiting using `slowapi` to restrict the number of requests.
- Benefits: Protects the service from being overwhelmed by too many requests.
- Trade-offs: Adds slight overhead to request processing.
- Bottlenecks: May need fine-tuning for different load scenarios.

9. PostgreSQL and SQLModel
- Achieved: Used PostgreSQL as the database and SQLModel as the ORM.

10. Data Validation
- Achieved: Added validation for "submitted procedure" and "Provider NPI".
- Benefits: Ensures data integrity and correctness.

11. Concurrent Processing
- Achieved:* Multiple instances of both claim_process and payments services can run concurrently.
- Benefits: Allows the system to handle a large volume of claims efficiently by distributing the load.
- Trade-offs: Requires proper load balancing and state management.
- Bottlenecks: Potential contention on shared resources like the database or message queue.

Partially achieved:
How will I solved if I have more time

Failure Handling and Concurrent Processing

Failure Handling
- Service Failure: If claim_process fails to publish a message to RabbitMQ, the failure is logged, and the service retries publishing the message. If the payments service fails to consume a message, it logs the error and retries consumption.
- Database Failure: If there is a database failure in claim_process, the transaction is rolled back to maintain data integrity. Similar rollback mechanisms are in place for the payments service.
- RabbitMQ Failure: If RabbitMQ is down, messages are not lost but retried when RabbitMQ is back up. Implementing dead-letter exchanges in RabbitMQ can help manage failed messages.

Steps to Unwind
1. Claim_Process Service Failure:
   - Log the error.
   - Retry publishing the message.

   - If persistent failure, send an alert to the operations team.
2. Payments Service Failure:
   - Log the error.
   - Retry message consumption.
   - If persistent failure, send an alert to the operations team.

Architecture

Components

1. **API Gateway**:
   - Handles incoming HTTP requests.
   - Implements rate limiting to ensure no more than 10 requests per minute for certain endpoints.

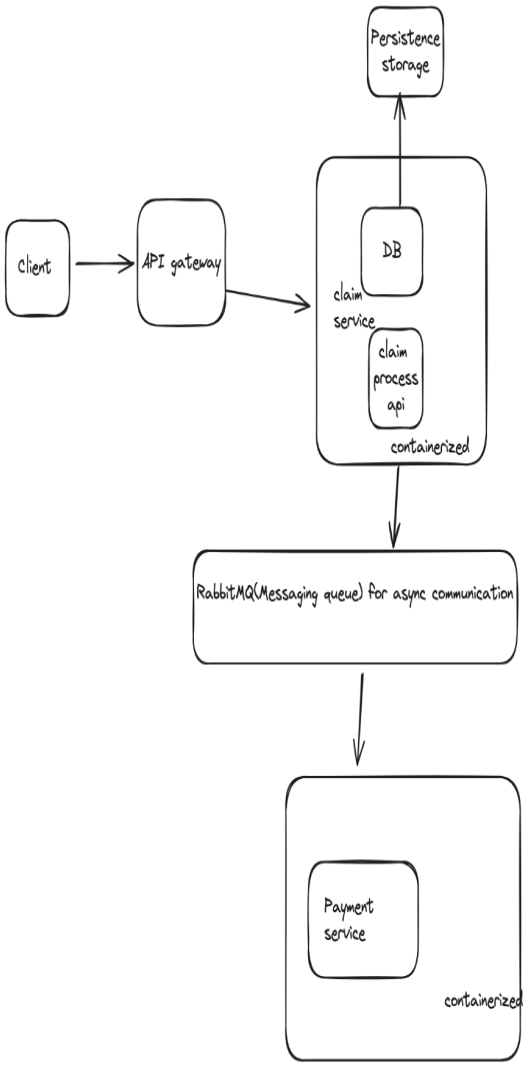
2. **Claim Process Service**:
   - Processes claim data from uploaded CSV files.
   - Computes the net fee for each claim.
   - Stores claims in PostgreSQL.
   - Publishes net fee data to RabbitMQ for consumption by the payments service.

3. **RabbitMQ**:
   - Acts as a message broker, decoupling producers (claim_process) and consumers (payments).

4. **Payments Service**:
   - Consumes net fee messages from RabbitMQ.
   - Processes and logs the received net fee data.

5. **PostgreSQL**:
   - Stores claim data.
   - Used by the claim_process service for data persistence.

Diagram



Implementation

Claim Process Service

1. **Endpoint for Processing Claims**:
   - Accepts CSV file uploads.
   - Parses the file and computes net fees.
   - Stores claims in PostgreSQL.
   - Publishes net fee data to RabbitMQ.

   ```python
   @router.post("/claims/")
   async def process_claims(file: UploadFile, db: AsyncSession = Depends(get_db)):
       claims_data = await parse_and_process_csv(file)
   ```

```
    for claim_data in claims_data:
        new_claim = await create_claim(db, claim_data)
        if new_claim is None:
            return {"error": "Duplicate claim detected"}
    return {"status": "claims processed successfully"}
```

2. **Top 10 Providers Endpoint**:
   - Returns the top 10 provider NPIs by net fees.

   ```python
   @router.get("/providers/top10", response_model=List[ProviderNetFee])
   async def read_top_10_providers(db: AsyncSession = Depends(get_db)):
       providers = await get_top_10_providers_by_net_fee(db)
       return providers
   ```

3. **Rate Limiting**:
   - Implemented using `slowapi`.

   ```python
   @limiter.limit("10/minute")
   ```

4. **Data Validation**:
   - Validate "submitted procedure" and "Provider NPI".

   ```python
   def validate_claim_data(row: Dict) -> bool:
       if not row["submitted_procedure"].startswith("D"):
           return False
       if not re.match(r"^\d{10}$", row["provider_npi"]):
           return False
       return True
   ```

Payments Service

1. **Message Consumer**:
   - Consumes messages from RabbitMQ.

- Logs and processes the received net fee data.

```python
async def consume_net_fee():
    connection = await aio_pika.connect_robust("amqp://guest:guest@rabbitmq/")
    async with connection:
        channel = await connection.channel()
        queue = await channel.declare_queue("net_fee", auto_delete=True)

        async for message in queue:
            async with message.process():
                net_fee_data = json.loads(message.body.decode())
                logger.info(f"Received net fee data: {net_fee_data}")
                print(f"Received net fee data: {net_fee_data}")
```

Database Setup
- **PostgreSQL Database**:
  - Stores claims data.
  - Configured using Docker Compose.

```yaml
services:
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB

: postgres
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - my_network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d postgres"]
      interval: 10s
```

```
        timeout: 5s
        retries: 5
    ```

Dockerization
- **Docker Compose**: Used to bring up all services (API Gateway, Claim Process, Payments, RabbitMQ, PostgreSQL).

    ```yaml
    version: '3.8'
    services:
      db:
        image: postgres:13
        environment:
          POSTGRES_USER: user
          POSTGRES_PASSWORD: password
          POSTGRES_DB: postgres
        ports:
          - "5432:5432"
        volumes:
          - postgres_data:/var/lib/postgresql/data
        networks:
          - my_network
        healthcheck:
          test: ["CMD-SHELL", "pg_isready -U user -d postgres"]
          interval: 10s
          timeout: 5s
          retries: 5
      rabbitmq:
        image: "rabbitmq:3-management"
        ports:
          - "5672:5672"
          - "15672:15672"
        networks:
          - my_network
        healthcheck:
          test: ["CMD-SHELL", "rabbitmq-diagnostics -q ping"]
          interval: 10s
          timeout: 5s
          retries: 5
```

```
  claim_process:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - .:/app
    networks:
      - my_network
    ports:
      - "8001:8000"
    depends_on:
      db:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    entrypoint: ["/bin/bash", "-c", "python /app/app/init_db_claim.py && uvicorn app.main:app
--host 0.0.0.0 --port 8000"]
  payments:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - .:/app
    networks:
      - my_network
    ports:
      - "8002:8001"
    depends_on:
      rabbitmq:
        condition: service_healthy
networks:
  my_network:
    external: true
volumes:
  postgres_data:
```

Communication with Payments Service

Pseudocode for Communication

Publish Net Fee to RabbitMQ:
```python
async def publish_net_fee(claim_data: Dict):
    try:
        logging.info("Connecting to RabbitMQ...")
        connection = await aio_pika.connect_robust("amqp://guest:guest@rabbitmq/")
        async with connection:
            logging.info(f"Connected to RabbitMQ. Publishing data: {claim_data}")
            channel = await connection.channel()
            await channel.default_exchange.publish(
                aio_pika.Message(body=json.dumps(claim_data).encode()),
                routing_key="net_fee",
            )
        logging.info("Data published to RabbitMQ.")
    except Exception as e:
        logging.error(f"Error publishing to RabbitMQ: {e}")
```

Consume Net Fee from RabbitMQ:
```python
async def consume_net_fee():
    while True:
        try:
            connection = await aio_pika.connect_robust("amqp://guest:guest@rabbitmq/")
            async with connection:
                channel = await connection.channel()
                queue = await channel.declare_queue("net_fee", auto_delete=True)
                async for message in queue:
                    async with message.process():
                        net_fee_data = json.loads(message.body.decode())
                        logger.info(f"Received net fee data: {net_fee_data}")
                        print(f"Received net fee data: {net_fee_data}")
        except aio_pika.exceptions.ConnectionClosed:
            logging.warning("RabbitMQ connection closed, retrying in 5 seconds...")
            await asyncio.sleep(5)
```