sahanasatishhh final commit:) 031f327 · 19 minutes ago History

Preview Code Blame Raw

[KG1] Endpoint Definitions

Knowledge Goal Number & Name

[KG1] Endpoint Definitions

File Reference

`user-service/app/main.py:120–330`

- more information is available in the README.md for the the entire endpoints and their definitions.

Code Fragments

```python
@app.post("/users", status_code=201)
async def create_user(user: UserCreate, request: Request):
    #creayes and store user data in Postgres

@app.get("/user-avail/cache-aside")
async def get_user_avail_cache_aside(
```

```
    request: Request,
    user1email: str = Query(...)
):
    #try and access the data in Redis or if nto present acces it in postgres if present, put it in Redis and t
```

Justification

These endpoints define unique URL paths that expose the functionality like creating users, caching accessed users for the user-service. They allow external clients and internal services to create users and retrieve their availability data through well-defined HTTP routes. It is the only service with a database and maintains the isolation.

[KG2] Containerization

Knowledge Goal Number & Name

[KG2] Containerization

File Reference

`user-service/Dockerfile:1-13`

Code Fragment

```
FROM python:3.11-slim

RUN apt-get update && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY requirements.txt .
```

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

COPY app/ ./app/

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

Justification

This Dockerfile defines the build process for the users-service microservice. By setting the base image, installing dependencies, and defining the startup command, it bundles the application code and environment into a single, isolated, and portable container image, which is the definition of Containerization.

Demonstrating Container Orchestration & API Gateway [KG3 & KG4]

Knowledge Goal Number & Name: [KG3] Container Orchestration & [KG4] API Gateway

File Reference: docker-compose.yml:101-120

Code Fragment

```
availability-service:
    build:
      context: ./availability-service
      dockerfile: Dockerfile
    environment:
      - USER_SERVICE_BASE=${USER_SERVICE_BASE}
    healthcheck:
      test: ["CMD-SHELL", "curl -fsS http://localhost:8000/health || exit 1"]
      interval: 60s
      timeout: 3s
```

```
      retries: 10
      start_period: 80s
    depends_on:
      user-service:
        condition: service_healthy
    restart: unless-stopped
    volumes:
      - ./availability-service:/app
    networks:
      - w2meet-network
```

Justification:

This docker-compose.yml file defines and manages the multi-service application, demonstrating Container Orchestration. The gateway service acts as the API Gateway, exposing port 80 to the outside world and routing requests to the backend services (users_service and posts_service) based on the depends_on setting and its internal configuration.

Demonstrating Load Balancing [KG5]

Knowledge Goal Number & Name: [KG5] Load Balancing File Reference: nginx/nginx.conf:1-16 Code Fragment:

```
Nginx
upstream user_service {
    server user-service:8000;
}

upstream availability_service {
    server availability-service:8000;
}
upstream suggestion_service {
    server suggestion-service:8000;
}
upstream worker_service {
```

```
    server worker-service:8000;
}

server {
    listen 80;
    # ... proxy pass to upstream ...
```

Justification: This Nginx configuration fragment defines an upstream block named post_servers that lists multiple, identical instances of the post-processing service. The Nginx server is configured to distribute incoming traffic across these defined servers, automatically implementing Load Balancing to ensure no single instance is overwhelmed and to improve system availability.

[KG6] Reliability, Scalability, and Maintainability

Knowledge Goal Number & Name

[KG6] Reliability, Scalability, Maintainability

File References

- availability-service/app/main.py:80-99, 172-175
- docker-compose.yml:101-138

Code Fragment 1: Resilient Error Handling with Proper HTTP Codes

```
    if user1_resp.status_code == 404 or user2_resp.status_code == 404:
        raise HTTPException(status_code=404, detail="One or both users not found")
    if user1_resp.status_code >= 400 or user2_resp.status_code >= 400:
        raise HTTPException(status_code=502, detail="User service error")
```

Justification

```
Services gracefully handle downstream failures and return correct HTTP status codes (404, 502) for
`availability-service`. This demonstrates reliability by preventing untraceable failures and giving clients
meaningful error responses.
```

Code Fragment 2: Service Isolation via Docker Compose (Scalability & Maintainability)

```yaml
  availability-service:
    build:
      context: ./availability-service
      dockerfile: Dockerfile
    environment:
      - USER_SERVICE_BASE=${USER_SERVICE_BASE}
    healthcheck:
      test: ["CMD-SHELL", "curl -fsS http://localhost:8000/health || exit 1"]
      interval: 60s
      timeout: 3s
      retries: 10
      start_period: 80s
    depends_on:
      user-service:
        condition: service_healthy
    restart: unless-stopped
    volumes:
      - ./availability-service:/app
    networks:
      - w2meet-network
  suggestion-service:
    build:
      context: ./suggestion-service
      dockerfile: Dockerfile
    environment:
      - AVAILABILITY_SERVICE_BASE=${AVAILABILITY_SERVICE_BASE}
```

```
    healthcheck:
      test: ["CMD-SHELL", "curl -fsS http://localhost:8000/health || exit 1"]
      interval: 90s
      timeout: 3s
      retries: 10
      start_period: 10s
    depends_on:
      availability-service:
        condition: service_healthy
    restart: unless-stopped
    volumes:
      - ./suggestion-service:/app

    networks:
      - w2meet-network
```

Justification

Each microservice is built and deployed as an independent container, enabling horizontal scalability by design. Although replicas are not explicitly configured, this structure allows services to be scaled independently in the future without code changes. Clear separation of services improves maintainability by isolating responsibilities.

[KG7] Service Communication

Knowledge Goal Number & Name

[KG7] Service Communication

File Reference

`availability-service/app/main.py:157–162`

Code Fragment - Distributed Read

```
async with httpx.AsyncClient(timeout=10.0) as client:
            user1_resp = await client.get(
                f"{USER_SERVICE_BASE}/user-avail/cache-aside",
                params={"user1email": userId1},
                headers={"Case-ID": case_id},
            )
```

CRUD operations accessible only via user-service through the API Gateway (enables inter-service communication for availability,suggestion and worker who wanna)

Justification: This fragment demonstrates **synchronous inter-service communication using HTTP REST**, because `availability-service` makes a blocking `GET` request to `user-service` (via `httpx.AsyncClient().get(...)`) to fetch each user's availabilities before it can compute the intersection. The request must complete successfully (or return an error like 404/503) before `availability-service` can respond, proving the dependency and synchronous workflow across services. This is done due to perform error handling with ease.

[KG8] Caching Strategy (Cache-Aside)

Knowledge Goal Number & Name

[KG8] Caching Strategy

File Reference

`user-service/app/main.py:140–190`

Code Fragment

```
cached_data = redis_client.get(f"cache_aside_{user1email.upper()}")
if cached_data:
            logging.info(f"[{case_id}] CACHE ASIDE: CACHE HIT with key cache_aside_{user1email.upper()} servec
            return json.loads(cached_data)
        else:
            logging.info(f"[{case_id}] CACHE ASIDE: CACHE MISS with key cache_aside_{user1email.upper()} fetch
            #default base is USD
            try:
                with engine.connect() as conn:
                    txt=text(f"SELECT * FROM USERAVAIL WHERE email='{user1email}'")
                    res=conn.execute(txt)
                    rows=res.fetchall()
    #Postgres continues to fetch and also propagate this data up redis for faster access
```

Justification

```
This implements a cache-aside strategy using Redis inside `user-service`. Frequently accessed data is
served from memory, reducing database load and improving latency.
```

[KG9] Observability (Logging & Tracing)

Knowledge Goal Number & Name

[KG9] Observability (Logging & Tracing)

File Reference

`availability-service/app/main.py:36-40`

Code Fragment

```
logger.info(f"[{case_id}] Request started - Method={request.method} Path={request.url.path}")
    #Pass the request forward to the next middleware in the nextservice chain
    response = await call_next(request)
    response.headers["Case-ID"] = case_id
    logger.info(f"[{case_id}] Request completed - Status={response.status_code}, , Time taken={(time.perf_cour
```

Justification

```
Structured logs with a propagated Case-ID enable tracing requests across services. This improves
debuggability and visibility into system behavior.
```

[KG11] Asynchronous Messaging

Knowledge Goal Number & Name

[KG11] Asynchronous Messaging

File Reference

`worker-service/app/main.py:201–206`

Code Fragment

```
message = aio_pika.Message(
        body=json.dumps(payload).encode("utf-8"),
        delivery_mode=aio_pika.DeliveryMode.PERSISTENT,
```

```
    )

await rmq_channel.default_exchange.publish(message, routing_key=QUEUE_NAME)
```

Justification

```
Jobs are published to RabbitMQ and processed asynchronously by worker services. This prevents long-running
tasks from blocking user-facing API requests and improves system responsiveness.

Example:
If many users request suggestions at the same time (e.g., during peak scheduling hours), jobs are queued in
RabbitMQ and processed by workers  asynchronously, allowing the API to respond immediately instead of
blocking while suggestions are computed. (Although only one worker instance is deployed in this demo, the
queue-based design enables horizontal scaling by simply starting additional worker containers).
```

Performance Impact (few notes)

```
- Caching reduced repeated availability lookups and directly contributed to improved response times
measured during load testing.
- Repeated tests on some error handling cases and the time taken in ms (synchronous) was about 41 ms for
one of the requests in the `test_error_handling.sh` compared to the usual 65 ms successful suggestion
service runs.
- Successful  Cache hits found in log aand it reduced the time to 2.5 ms for each subsequent access in
user-service!
```