

Wahyu Fadilah Saraswati

11221024

Sistem Paralel dan Terdistribusi - B

Ujian Tengah Semester

Pub-Sub Log Aggregator dengan Idempotent Consumer dan Deduplication

1. Jawab ringkas-padat (sekitar 150–250 kata per poin) dan sertakan sitasi ke bab terkait di buku utama. Gunakan istilah teknis dalam Bahasa Inggris bila relevan.

- a. T1 (Bab 1): Jelaskan karakteristik utama sistem terdistribusi dan trade-off yang umum pada desain Pub-Sub log aggregator.

Menurut Coulouris et al. (2012), sistem terdistribusi adalah sistem di mana *hardware* atau *software components* pada komputer-komputer yang saling terhubung berkomunikasi dan berkoordinasi hanya melalui pengiriman pesan (*message passing*). Sistem ini memiliki tiga karakteristik utama, yaitu *Concurrency*, *Lack of Global Clock*, dan *Independent Failures*. *Concurrency* berarti beberapa proses berjalan paralel di komputer berbeda untuk efisiensi dan skalabilitas, namun menimbulkan tantangan koordinasi dan konsistensi data. *Lack of Global Clock* menunjukkan tidak adanya jam global karena keterbatasan sinkronisasi antar komputer, sehingga penting dalam pengurutan peristiwa (*event ordering*). Sementara itu, *Independent Failures* menggambarkan setiap komponen dapat gagal sendiri tanpa menghentikan yang lain, menuntut desain sistem yang *fault-tolerant* (Coulouris et al., 2012).

Menurut Tanenbaum & van Steen (2023), *trade-off* dalam sistem terdistribusi adalah kompromi antara tujuan desain seperti *performance*, *reliability*, dan *consistency*. Coulouris et al. (2012) menambahkan bahwa peningkatan satu aspek dapat menurunkan aspek lain. Dalam desain *Publish-Subscribe (Pub-Sub) log aggregator*, *trade-off* ini muncul antara kinerja, keandalan, konsistensi, dan skalabilitas. Sistem Pub-Sub yang *loosely coupled* memprioritaskan skalabilitas dan kinerja dengan memilih *eventual consistency*, tetapi kehilangan *strong consistency*. Sebaliknya, menambah mekanisme *message persistence* dan *acknowledgment* meningkatkan keandalan namun memperlambat sistem. Desain Pub-Sub log aggregator merepresentasikan keseimbangan antara efisiensi, konsistensi, dan ketahanan yang menjadi inti dari sistem terdistribusi modern.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Bab 1, hal. 2, 16–17).

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Vrije Universiteit Amsterdam. (Bab 1, hal. 9-10).

- b. T2 (Bab 2): Bandingkan arsitektur client-server vs publish-subscribe untuk aggregator. Kapan memilih Pub-Sub? Berikan alasan teknis.

Coulouris et al. (2012) menjelaskan bahwa arsitektur *client-server* merupakan model komunikasi terpusat di mana *client* mengirimkan permintaan (*request*) dan *server* memberikan tanggapan (*response*). Model ini umum digunakan, namun bersifat *tightly coupled*, artinya komponen *client* dan *server* saling bergantung erat—*client* harus mengetahui lokasi dan ketersediaan server untuk bisa berkomunikasi. Pendekatan ini sesuai untuk sistem dengan interaksi sinkron dan jumlah klien terbatas, tetapi kurang efisien ketika beban sistem meningkat karena seluruh pemrosesan terpusat pada satu titik.

Sebaliknya, arsitektur *Publish-Subscribe (Pub-Sub)* bersifat *loosely coupled*, artinya *publishers* (pengirim data) dan *subscribers* (penerima data) tidak saling mengetahui identitas satu sama lain (Coulouris et al., 2012). *Publishers* mengirimkan data atau peristiwa (*events*) tanpa menentukan penerimanya, sementara *subscribers* menerima data yang relevan melalui perantara (*broker*). Menurut Tanenbaum & van Steen (2023), desain ini memungkinkan *asynchronous communication*, yakni proses komunikasi yang tidak memerlukan respons segera, sehingga sistem dapat tetap berjalan meskipun salah satu komponen sedang tidak aktif.

Untuk sistem *log aggregator*, *Pub-Sub* lebih sesuai karena mampu menangani arus data besar dari berbagai sumber secara bersamaan dengan *high throughput* (kemampuan memproses data berkecepatan tinggi) dan *fault tolerance* (kemampuan bertahan ketika sebagian node gagal). Selain itu, sifatnya yang berbasis *event-driven processing* (pemrosesan data dipicu oleh kedatangan *event* baru) memungkinkan pembaruan log secara real-time tanpa ketergantungan langsung antara pengirim dan penerima.

Publish-subscribe dipilih untuk sistem agregasi log berskala besar yang memerlukan komunikasi asinkron, skalabilitas tinggi, dan ketahanan terhadap kegagalan. Sementara itu, *client-server* lebih cocok untuk transaksi langsung yang terpusat, di mana setiap permintaan memiliki tanggapan yang pasti dari satu server.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Bab 2, hal. 47–48, 60)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Bab 2, hal. 68–70)

- c. T3 (Bab 3): Uraikan at-least-once vs exactly-once delivery semantics. Mengapa idempotent consumer krusial di presence of retries?

Dalam sistem terdistribusi, *delivery semantics* mengacu pada jaminan mengenai berapa kali suatu pesan atau operasi dijalankan oleh penerima. Menurut Coulouris et al. (2012), *at-least-once semantics* berarti permintaan atau pesan dijamin dijalankan minimal satu kali di sisi

server. Hal ini biasanya dicapai melalui mekanisme *retransmission* (pengiriman ulang) ketika klien tidak menerima respons, namun akibatnya operasi yang sama bisa dieksekusi lebih dari sekali. Sebaliknya, *exactly-once semantics* menjamin bahwa setiap operasi dijalankan tepat satu kali, tidak lebih dan tidak kurang, sehingga hasil yang diterima oleh klien mencerminkan satu eksekusi tunggal dari prosedur tersebut. Meskipun demikian, pencapaian *exactly-once semantics* membutuhkan deteksi duplikasi dan pencatatan hasil (*result logging*) agar server tidak mengeksekusi kembali permintaan yang sama (Coulouris et al., 2011).

Menurut Tanenbaum & van Steen (2023) menambahkan bahwa *at-least-once semantics* memberikan kepastian bahwa pemanggilan jarak jauh (*remote procedure call*) telah dilakukan setidaknya sekali, tetapi tidak dapat menjamin bahwa eksekusi tidak terjadi berulang. Kondisi ideal yaitu *exactly-once semantics* sulit dicapai karena adanya kemungkinan *crash* dan kehilangan pesan dalam jaringan. Oleh karena itu, dibutuhkan *idempotent consumer*, yaitu penerima yang tetap menghasilkan hasil yang sama walaupun menerima pesan yang sama berulang kali. *Idempotent* berarti operasi yang diulang akan memberikan efek yang identik dengan satu kali eksekusi. Konsumen yang idempotent krusial dalam kondisi *retries* (pengulangan permintaan akibat kegagalan) agar sistem tetap konsisten dan tidak menghasilkan efek ganda, misalnya pada transaksi keuangan atau pembaruan data yang sensitif terhadap duplikasi.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Hal. 198-200)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Hal. 511)

- d. T4 (Bab 4): Rancang skema penamaan untuk `topic` dan `event_id` (unik, collision-resistant). Jelaskan dampaknya terhadap dedup.

Dalam sistem terdistribusi, *naming* (penamaan) merupakan mekanisme penting untuk mengidentifikasi entitas seperti *topic* dan *event* secara unik. Menurut Coulouris et al. (2012), skema penamaan yang baik harus mendukung *uniqueness* (keunikan) dan *collision resistance* (ketahanan terhadap tabrakan nama). Dalam konteks *event-based system*, sebuah *topic* dapat diberi nama secara hierarkis, misalnya `/system/sensor/temperature` atau `/finance/stock/updates`. Struktur hierarki ini membantu proses *name resolution* (pemecahan nama) dan mendukung skala besar dengan meminimalkan konflik antar topik.

Selain itu, setiap peristiwa (*event*) perlu memiliki `event_id` yang bersifat unik dan tahan terhadap *collision*. Coulouris et al. (2012) menyarankan penggunaan *hash function* seperti SHA-256 untuk membangkitkan pengenal berbasis konten atau waktu, misalnya `event_id = hash(topic+timestamp+source_id)`. Dengan skema ini, setiap peristiwa dijamin memiliki identitas yang unik walaupun berasal dari topik yang sama.

Van Steen dan Tanenbaum (2023) menegaskan bahwa *identifier* yang baik harus memenuhi tiga kriteria, yaitu merujuk ke satu entitas saja, tidak digunakan ulang, dan selalu mengacu pada entitas yang sama sepanjang waktu. Penggunaan pengenal berbasis hash atau UUID (*Universally Unique Identifier*) membantu memastikan sifat tersebut.

Dari sisi *deduplication* (penghapusan duplikat), skema penamaan yang unik dan tahan tabrakan memungkinkan sistem mendeteksi dan mengabaikan *event* yang sama yang dikirim ulang. Ketika sistem memeriksa *event_id* dan menemukan bahwa pengenal tersebut telah ada, maka pesan tidak akan diproses ulang, sehingga meningkatkan konsistensi data serta efisiensi komunikasi.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Hal. 161-164)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Hal. 326-329)

- e. T5 (Bab 5): Bahas ordering: kapan total ordering tidak diperlukan? Usulkan pendekatan praktis (mis. event timestamp + monotonic counter) dan batasannya.

Dalam sistem terdistribusi, *ordering* atau pengurutan peristiwa berfungsi untuk menentukan urutan terjadinya *event* antar proses agar sistem tetap konsisten. Dalam banyak situasi, proses dapat berjalan secara independen tanpa keterkaitan sebab-akibat (*causal relation*), sehingga penerapan *total ordering* justru menambah kompleksitas sistem dan overhead komunikasi.

Menurut Tanenbaum dan van Steen (2023), jika dua proses tidak saling berinteraksi, maka urutan global atau *total order* tidak diperlukan karena tidak ada hubungan sebab-akibat di antara mereka. Pandangan serupa dijelaskan oleh Coulouris et al. (2012), yang menyatakan bahwa *causal ordering* dan *FIFO ordering* hanya menghasilkan *partial ordering*, dan *total ordering* hanya diperlukan jika seluruh proses harus melihat semua kejadian dalam urutan yang sama. Dengan demikian, ketika *event* bersifat independen atau *concurrent* (terjadi bersamaan), sistem cukup menggunakan *partial ordering* (urutan sebagian) seperti *FIFO* atau *causal order*. Contoh penerapan yang tidak membutuhkan *total ordering* adalah sistem papan buletin seperti USENET, karena urutan pesan tidak selalu penting bagi setiap pengguna (Coulouris et al., 2012).

Pendekatan praktis yang umum digunakan adalah kombinasi *event timestamp* (penanda waktu peristiwa) dan *monotonic counter* (penghitung monotonik). Setiap proses memiliki penghitung lokal (*counter*) yang meningkat setiap kali terjadi *event*, dan nilai penghitung tersebut disertakan dalam pesan sebagai *timestamp*. Metode ini dikenal sebagai *logical clock*, yang menjaga urutan sebab-akibat tanpa perlu sinkronisasi waktu fisik (Coulouris et al., 2012). Meski efisien, pendekatan ini memiliki batasan, yaitu tidak dapat membedakan *concurrent events*, tidak

menjamin urutan global, dan memerlukan mekanisme tambahan seperti *vector clocks* atau *sequencer* (penyusun urutan) untuk mencapai *total order*.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Bab 14, hal. 609. Bab 15, hal. 651-653)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Bab 5, Hal. 260-267)

- f. T6 (Bab 6): Identifikasi failure modes (duplikasi, out-of-order, crash). Jelaskan strategi mitigasi (retry, backoff, durable dedup store).

Dalam sistem terdistribusi, terdapat beberapa *failure modes* (pola kegagalan) yang umum terjadi, yaitu *crash*, *duplication*, dan *out-of-order delivery*. *Crash failure* terjadi ketika suatu proses berhenti secara tiba-tiba dan tidak lagi menanggapi permintaan dari komponen lain. Menurut van Steen dan Tanenbaum (2023), *crash failure* adalah kondisi di mana suatu server berhenti sebelum waktunya tetapi berfungsi dengan benar hingga proses tersebut berhenti. Kondisi ini dapat menyebabkan layanan menjadi tidak responsif dan mengganggu alur komunikasi antar proses.

Van Steen dan Tanenbaum (2023) menjelaskan bahwa duplikasi pesan merupakan bentuk *arbitrary failure* (kegagalan tak terduga yang dapat terjadi dalam berbagai bentuk, misalnya pesan duplikat atau pesan rusak) yang dapat muncul karena pesan tersimpan di buffer jaringan dalam waktu yang lama. Sedangkan *out-of-order delivery* muncul ketika pesan diterima tidak sesuai urutan pengirimannya, yang dalam beberapa aplikasi dianggap sebagai bentuk kegagalan karena pesan diharapkan tiba sesuai urutan. Coulouris et al. (2012) menyebut bahwa beberapa aplikasi menuntut pesan dikirim dalam urutan pengirim, sehingga pesan yang datang tidak sesuai urutan dianggap gagal.

Untuk memitigasi kegagalan tersebut, strategi yang umum digunakan adalah *retry*, *backoff*, dan *durable dedup store*. Strategi *retry* dilakukan dengan mengirim ulang permintaan setelah kegagalan sementara, sementara *exponential backoff* menunda pengiriman ulang dengan waktu tunda yang meningkat agar jaringan tidak padat (van Steen & Tanenbaum, 2023). Selain itu, *durable dedup store* digunakan untuk mendeteksi dan menolak duplikasi dengan menyimpan identitas unik dari setiap permintaan, sedangkan mekanisme *message sequencing* dan *acknowledgment* memastikan pesan diproses sesuai urutannya (Coulouris et al., 2012).

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Bab 4, hal. 149)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Bab 8, hal. 467, 508, 513, 536-538)

- g. T7 (Bab 7): Definisikan eventual consistency pada aggregator; jelaskan bagaimana idempotency + dedup membantu mencapai konsistensi.

Eventual consistency (konsistensi akhir) pada *aggregator* (layanan pengumpul data) adalah jaminan bahwa *aggregator* pada akhirnya akan memiliki data yang akurat dan lengkap dari semua sumber yang dikumpulkannya, meskipun data tersebut tiba secara tertunda, tidak berurutan, atau terduplikasi. Van Steen dan Tanenbaum (2023) mendefinisikan *eventual consistency* sebagai kondisi di mana, jika tidak ada pembaruan (*updates*) yang terjadi untuk waktu yang lama, semua replika secara bertahap akan menjadi konsisten. Sistem semacam ini tidak menuntut keseragaman instan, tetapi menjamin bahwa setelah periode tertentu tanpa pembaruan tambahan, semua data agregat akan sinkron.

Agar *eventual consistency* dapat tercapai, sistem perlu menerapkan *idempotency* (idempoten) dan *deduplication* (penghilangan duplikasi). *Idempotency* memastikan bahwa operasi yang dijalankan berulang kali tetap menghasilkan hasil akhir yang sama, sehingga pembaruan ganda akibat retransmisi tidak menyebabkan inkonsistensi. *Deduplication* adalah mekanisme teknis untuk mencapai *idempotency* tersebut, biasanya dengan cara *aggregator* melacak ID unik dari pesan yang sudah diproses dan mengabaikan duplikat yang masuk kemudian. Van Steen dan Tanenbaum (2023) menekankan bahwa kedua mekanisme ini berperan penting dalam replikasi asinkron karena membantu menjaga konvergensi, yaitu keadaan di mana semua replika mencapai nilai yang sama setelah proses sinkronisasi selesai. Kombinasi *eventual consistency*, *idempotency*, dan *deduplication* membuat *aggregator* tetap andal (*reliable*) dan konsisten meskipun menghadapi duplikasi pesan maupun keterlambatan propagasi data.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley. (Bab 13, hal. 574)

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology. (Bab 7)

- h. T8 (Bab 1–7): Rumuskan metrik evaluasi sistem (throughput, latency, duplicate rate) dan kaitkan ke keputusan desain.

Dalam sistem terdistribusi, kinerja sistem dapat dievaluasi menggunakan beberapa metrik utama, yaitu *throughput*, *latency*, dan *duplicate rate*. *Throughput* (laju pemrosesan) mengukur seberapa banyak permintaan atau operasi yang dapat diselesaikan oleh sistem dalam satuan waktu tertentu. Metrik ini mencerminkan efisiensi sistem dalam memanfaatkan sumber daya dan menangani beban kerja. Menurut Coulouris et al. (2012), kinerja sistem terdistribusi umumnya diukur dari *responsiveness* (ketanggapan) dan *throughput*, karena keduanya menunjukkan laju dan kualitas layanan yang diberikan. *Latency* (waktu tunda) mengacu pada selang waktu antara pengiriman permintaan oleh klien hingga penerimaan respons dari server.

Coulouris et al. (2012) menjelaskan bahwa *latency* bergantung pada waktu transmisi jaringan, penjadwalan proses, serta kecepatan eksekusi pada sisi server.

Sementara itu, *duplicate rate* (tingkat duplikasi pesan) digunakan untuk mengukur frekuensi terjadinya pesan yang dikirim atau diterima lebih dari satu kali akibat mekanisme retransmisi atau ketidakandalan jaringan. Van Steen dan Tanenbaum (2023) menyebutkan bahwa pesan duplikat dapat terjadi pada saluran komunikasi yang tidak andal dan berdampak pada efisiensi sistem karena menambah beban jaringan dan menurunkan *throughput*. Ketiga metrik ini berpengaruh langsung terhadap keputusan desain sistem. Untuk meningkatkan *throughput*, sistem dapat menggunakan *replication* (replikasi) dan *load balancing* (pembagian beban) agar pemrosesan dapat dilakukan secara paralel di beberapa node (Coulouris et al., 2012). Untuk menurunkan *latency*, diterapkan *caching* (penyimpanan sementara) dan *data locality* (penempatan data yang dekat dengan pengguna). Sedangkan untuk menekan *duplicate rate*, digunakan *idempotency* (operasi yang hasilnya tidak berubah meski dijalankan berulang) serta *deduplication* (penghilangan duplikasi).

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley.

Tanenbaum, A. S., & van Steen, M. (2023). *Distributed Systems* (4th ed.). Delft University of Technology.

2. Bangun layanan aggregator berbasis Python (disarankan FastAPI/Flask + asyncio) dengan spesifikasi berikut:

- a. Model Event & API

- Event JSON minimal: { "topic": "string", "event_id": "string-unik", "timestamp": "ISO8601", "source": "string", "payload": { ... } }

src\models.py

```
from pydantic import BaseModel, Field
from datetime import datetime
from typing import Dict, List, Union

class Event(BaseModel):
    topic: str = Field(..., example="sensorA")
    event_id: str = Field(..., example="abc-123")
    timestamp: datetime
    source: str
    payload: Dict
```

```
class Stats(BaseModel):
    received: int
    unique_processed: int
    duplicate_dropped: int
    topics: List[str]
    uptime: str
```

Model **Event** didefinisikan menggunakan **Pydantic (FastAPI)** untuk melakukan validasi otomatis terhadap setiap data event yang diterima. *Pydantic* adalah library di FastAPI yang memvalidasi data otomatis berdasarkan tipe data Python. Jadi jika event JSON tidak sesuai format (misalnya timestamp tidak valid), server langsung menolak permintaan.

Struktur JSON yang digunakan sesuai spesifikasi tugas, yaitu berisi atribut **topic**, **event_id**, **timestamp**, **source**, dan **payload**. Tipe data **timestamp** mengikuti format **ISO8601** agar dapat dibaca secara universal oleh sistem. Model ini memastikan setiap event memiliki *unique identifier* (**event_id**) dan topik tertentu sebelum dimasukkan ke antrian pemrosesan.

- Endpoint **POST /publish** menerima batch atau single event; validasi skema.

src\api.py

```
from fastapi import APIRouter, BackgroundTasks, Query
from typing import List, Union
from src.models import Event, Stats
from src.metrics import metrics

router = APIRouter()

_events = []

@router.post("/publish")
async def publish(events: Union[Event, List[Event]],
background_tasks: BackgroundTasks, queue=None):
    if isinstance(events, Event):
        events = [events]
    background_tasks.add_task(publish_to_queue, events, queue)
    return {"accepted": len(events)}
```

Endpoint **POST /publish** digunakan oleh *publisher* untuk mengirimkan data event ke sistem aggregator. Endpoint ini mendukung dua format input, yaitu **single event** untuk satu

objek JSON dan **batch event** untuk daftar objek JSON. Validasi otomatis dilakukan berdasarkan model **Event**. Jika format data valid, setiap event akan dimasukkan ke *in-memory queue* secara asinkron melalui fungsi `publish_to_queue()` menggunakan mekanisme **BackgroundTasks** FastAPI. Setiap event yang valid akan dikirim ke antrian untuk diproses oleh *consumer*. Jika ada data yang tidak sesuai format JSON, FastAPI akan menolak dan menampilkan pesan kesalahan validasi. Hal ini membuat API tetap non-blocking, server tetap responsif walau ada banyak event yang dikirim bersamaan.

Fungsi `publish_to_queue()` menambahkan setiap event yang diterima ke dalam daftar `_events` untuk menyimpan data sementara dan **asyncio.Queue** (internal queue) untuk diproses oleh *consumer*. Pendekatan ini memisahkan lapisan penerimaan event (API) dengan lapisan pemrosesan event (*consumer*), sehingga sistem bisa bekerja asinkron dan tidak kehilangan event ketika trafik tinggi.

- Consumer (subscriber) memproses event dari internal queue (in-memory) dan melakukan dedup berdasarkan `(topic, event_id)`.

src\consumer.py

```
import asyncio, logging, json
from src.metrics import metrics

async def consumer_loop(queue, dedup_store):
    while True:
        event = await queue.get()
        metrics.received += 1
        metrics.topics.add(event.topic)

        is_new = await dedup_store.mark_processed(event.topic,
event.event_id)
        if is_new:
            metrics.unique_processed += 1
            logging.info(f"Processed:
{event.topic}/{event.event_id}")
        else:
            metrics.duplicate_dropped += 1
            logging.warning(f"Duplicate dropped:
{event.topic}/{event.event_id}")
        queue.task_done()
```

Komponen *consumer* berfungsi memproses event yang masuk dari *internal queue* berbasis **asyncio.Queue**. Setiap event yang diterima akan diperiksa ke **dedup store** (basis data SQLite) menggunakan kombinasi (**topic**, **event_id**) untuk mendeteksi duplikasi. *Deduplication* adalah proses menghindari pemrosesan berulang terhadap event yang sama. *In-memory queue* memastikan event tetap tertampung walaupun pemrosesan berjalan asynchronous. Jika event sudah pernah diproses sebelumnya, sistem tidak akan memproses ulang dan hanya mencatat log duplikasi. Jika event baru, sistem akan memprosesnya dan menandai event tersebut di dedup store.

- Endpoint **GET /events?topic=...** mengembalikan daftar event unik yang telah diproses.

src\api.py

```
from fastapi import APIRouter, BackgroundTasks, Query
from typing import List, Union
from src.models import Event, Stats
from src.metrics import metrics

router = APIRouter()

_events = []

@router.get("/events")
async def get_events(topic: str = Query(None)):
    if topic:
        return [e for e in _events if e["topic"] == topic]
    return _events

async def publish_to_queue(events, queue):
    for e in events:
        _events.append(e.dict())
        await queue.put(e)
```

Endpoint ini digunakan untuk menampilkan daftar event yang sudah berhasil diproses oleh sistem. Parameter opsional **topic** dapat digunakan untuk memfilter event berdasarkan topik tertentu. Semua event yang dikembalikan berasal dari data unik yang tersimpan di dedup store. Endpoint ini berguna bagi *subscriber* lain untuk membaca hasil agregasi dari sistem. Jika **topic** tidak diberikan, maka semua event unik dari berbagai topik akan ditampilkan.

- Endpoint `GET /stats` menampilkan: `received`, `unique_processed`, `duplicate_dropped`, `topics`, `uptime`.

src\api.py

```
from fastapi import APIRouter, BackgroundTasks, Query
from typing import List, Union
from src.models import Event, Stats
from src.metrics import metrics

router = APIRouter()

@router.get("/stats", response_model=Stats)
async def get_stats():
    return {
        "received": metrics.received,
        "unique_processed": metrics.unique_processed,
        "duplicate_dropped": metrics.duplicate_dropped,
        "topics": list(metrics.topics),
        "uptime": metrics.uptime()
    }
```

Endpoint ini memberikan *runtime statistics* dari sistem aggregator, mencakup jumlah event yang diterima, jumlah event unik yang berhasil diproses, jumlah duplikasi yang terdeteksi dan dibuang, daftar topik yang aktif, serta lama waktu sistem berjalan (*uptime*). Informasi ini penting untuk mengevaluasi performa sistem secara real-time, terutama untuk mengukur *throughput* (jumlah event yang berhasil diproses per detik) dan *latency* (waktu tanggap sistem).

```
(venv) PS D:\uts-aggregator> Invoke-RestMethod -Uri "http://localhost:8080/stats"

received      : 0
unique_processed : 0
duplicate_dropped : 0
topics        : {}
uptime        : 532s
```

Statistik ini digunakan nanti dalam bagian evaluasi performa untuk mengukur seberapa cepat dan konsisten aggregator bekerja di bawah beban besar.

b. Idempotency & Deduplication

- Implementasikan dedup store yang tahan terhadap restart (contoh: SQLite atau file-based key-value) dan local-only.

src\dedup_store.py

```
import aiosqlite, datetime

class DedupStore:
    def __init__(self, path="dedup_store.db"):
        self.path = path
        self.db = None

    async def init(self):
        self.db = await aiosqlite.connect(self.path)
        await self.db.execute("PRAGMA journal_mode=WAL;")
        await self.db.execute("""
            CREATE TABLE IF NOT EXISTS processed_events (
                topic TEXT,
                event_id TEXT,
                processed_at TEXT,
                PRIMARY KEY (topic, event_id)
            )
        """)
        await self.db.commit()

    async def mark_processed(self, topic, event_id):
        ts = datetime.datetime.utcnow().isoformat()
        try:
            await self.db.execute(
                "INSERT INTO processed_events VALUES (?, ?, ?)",
                (topic, event_id, ts)
            )
            await self.db.commit()
            return True # baru pertama
        except aiosqlite.IntegrityError:
            return False # duplikat
```

Dedup store adalah komponen yang berfungsi untuk menyimpan riwayat event yang telah diproses, agar sistem dapat mengenali jika ada event yang sama dikirim ulang. Implementasi

di atas menggunakan SQLite (*aiosqlite*), *aiosqlite* digunakan karena mendukung operasi *asynchronous*, menjaga sistem tetap responsif walau melakukan I/O ke database. Database tersebut lokal, ringan dan secara otomatis menyimpan data di file `dedup_store.db`. Dengan desain ini, meskipun sistem dimatikan (*crash* atau *restart*), data dedup masih tersimpan, sehingga event yang sama tidak akan diproses dua kali. Hal ini memastikan sifat tahan terhadap restart (*crash-tolerant*) dan menjamin konsistensi hasil pemrosesan.

- Idempotency: satu event dengan (`topic`, `event_id`) yang sama hanya diproses sekali meski diterima berkali-kali. Dan
- Logging yang jelas untuk setiap duplikasi yang terdeteksi.

src\consumer.py

```
import asyncio, logging, json
from src.metrics import metrics

async def consumer_loop(queue, dedup_store):
    while True:
        event = await queue.get()
        metrics.received += 1
        metrics.topics.add(event.topic)

        is_new = await dedup_store.mark_processed(event.topic,
event.event_id)
        if is_new:
            metrics.unique_processed += 1
            logging.info(f"Processed:
{event.topic}/{event.event_id}")
        else:
            metrics.duplicate_dropped += 1
            logging.warning(f"Duplicate dropped:
{event.topic}/{event.event_id}")
        queue.task_done()
```

Idempotency memastikan bahwa operasi yang dijalankan berulang kali tetap menghasilkan hasil akhir yang sama. *Idempotency* sangat penting dalam sistem *at-least-once delivery*, karena pengiriman ulang bisa terjadi saat ada gangguan jaringan atau *publisher crash*. Pada implementasi di atas, setiap event yang masuk akan dicek ke dedup store melalui fungsi `mark_processed()`. Jika event belum ada di database, maka dianggap baru dan

akan diproses. Sedangkan jika *event* sudah pernah disimpan sebelumnya, sistem melewati pemrosesan agar tidak menghasilkan duplikasi efek.

```
PS D:\uts-aggregator> Invoke-RestMethod -Uri "http://localhost:8000/publish" -Method POST -Headers @{"Content-Type":"application/json"} -Body '{"topic":"demo","event_id":"1","timestamp":"2025-10-23T00:00:00Z","source":"test","payload":{"msg":"Hello"}}'

accepted
-----
1
Invoke-RestMethod -Uri "http://localhost:8000/publish" -Method POST -Headers @{"Content-Type":"application/json"} -Body '{"topic":"demo","event_id":"1","timestamp":"2025-10-23T00:00:00Z","source":"test","payload":{"msg":"Hello"}}'

accepted
-----
1
```

```
PS D:\uts-aggregator> .\venv\Scripts\activate
(venv) PS D:\uts-aggregator> uvicorn src.main:app --reload
INFO: Will watch for changes in these directories: ['D:\\uts-aggregator']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [1824] using WatchFiles
INFO: Started server process [26076]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:52630 - "GET / HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:51935 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:50547 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:50547 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:63097 - "POST /publish HTTP/1.1" 200 OK
00:59:43 - INFO - Processed: demo/1
INFO: 127.0.0.1:62545 - "POST /publish HTTP/1.1" 200 OK
01:02:44 - WARNING - Duplicate dropped: demo/1
```

Sistem mencatat log yang jelas untuk setiap event, baik baru maupun duplikat. Log ini digunakan untuk membantu proses debugging dan verifikasi bahwa deduplication berjalan dengan benar. Contoh hasil log di atas menunjukkan bahwa event pertama (*event_id*=1) diproses saat pertama kali dikirim, tetapi pada pengiriman kedua, sistem mengenalinya sebagai duplikat dan tidak memproses ulang. Hal ini membuktikan bahwa sistem memiliki mekanisme *idempotency* dan *deduplication* yang berfungsi sesuai rancangan.

c. Reliability & Ordering

- At-least-once delivery: simulasi duplicate delivery di publisher (mengirim beberapa event dengan *event_id* sama).

At-least-once delivery berarti setiap event dikirim minimal satu kali, tetapi bisa terkirim lebih dari satu kali akibat retry atau kegagalan jaringan. Implementasi aggregator menggunakan *deduplication* untuk menangani kondisi ini.

```
PS D:\uts-aggregator> Invoke-RestMethod -Uri "http://localhost:8000/publish" -Method POST -Headers @{"Content-Type":"application/json"} -Body '{"topic":"demo","event_id":"1","timestamp":"2025-10-23T00:00:00Z","source":"test","payload":{"msg":"Hello"}}'

accepted
-----
1
Invoke-RestMethod -Uri "http://localhost:8000/publish" -Method POST -Headers @{"Content-Type":"application/json"} -Body '{"topic":"demo","event_id":"1","timestamp":"2025-10-23T00:00:00Z","source":"test","payload":{"msg":"Hello"}}'

accepted
-----
1
```

Saat event dengan (`topic`, `event_id`) yang sama dikirim berulang, sistem tetap menerima semua (sesuai prinsip *at-least-once*), tetapi hanya memproses sekali saja, karena duplikasi dikenali dan dilewati.

```
PS D:\uts-aggregator> .\venv\Scripts\activate
(venv) PS D:\uts-aggregator> uvicorn src.main:app --reload
INFO: Will watch for changes in these directories: ['D:\uts-aggregator']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [1824] using WatchFiles
INFO: Started server process [26076]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:52630 - "GET / HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:51935 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:50547 - "GET /docs HTTP/1.1" 200 OK
INFO: 127.0.0.1:50547 - "GET /openapi.json HTTP/1.1" 200 OK
INFO: 127.0.0.1:63097 - "POST /publish HTTP/1.1" 200 OK
00:59:43 - INFO - Processed: demo/1
INFO: 127.0.0.1:62545 - "POST /publish HTTP/1.1" 200 OK
01:02:44 - WARNING - Duplicate dropped: demo/1
```

Hasil pengujian menunjukkan bahwa event pertama diproses (`[NEW]`), sedangkan event kedua diabaikan (`[DUPLICATE]`), menandakan bahwa sistem berhasil menangani duplikasi pengiriman.

- Toleransi crash: setelah restart container, dedup store tetap mencegah reprocessing event yang sama.

```
src\dedup_store.py
```

```
class DedupStore:
    def __init__(self, path="dedup_store.db"):
```

```
self.path = path
```

```
PS D:\uts-aggregator> Invoke-RestMethod -Uri "http://localhost:8000/publish" -Method POST -Headers @{"Content-Type"="application/json"} -Body '{"topic":"crash-test","event_id":"restart-001","timestamp":"2025-10-23T00:00:00Z","source":"test","payload":{"msg":"Before restart"}}'
```

```
accepted
-----
1
```

```
(venv) PS D:\uts-aggregator> uvicorn src.main:app --reload
INFO: Will watch for changes in these directories: ['D:\\uts-aggregator']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [22148] using WatchFiles
INFO: Started server process [21412]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:61747 - "POST /publish HTTP/1.1" 200 OK
01:49:09 - INFO - Processed: crash-test/restart-001
```

```
INFO: Shutting down
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: Finished server process [21412]
```

Sistem aggregator menggunakan **SQLite database (dedup_store.db)** sebagai dedup store. Karena data tersimpan di file lokal, informasi tentang event yang sudah diproses tetap ada meskipun sistem dimatikan atau container di-restart.

```
(venv) PS D:\uts-aggregator> uvicorn src.main:app --reload
INFO: Will watch for changes in these directories: ['D:\\uts-aggregator']
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO: Started reloader process [15216] using WatchFiles
INFO: Started server process [21148]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 127.0.0.1:49683 - "POST /publish HTTP/1.1" 200 OK
01:55:29 - WARNING - Duplicate dropped: crash-test/restart-001
```

Setelah server dijalankan ulang, event dengan (`topic`, `event_id`) yang sama dikirim ulang dan hasil log menunjukkan bahwa sistem mendeteksi event tersebut sebagai duplikat dan tidak memproses ulang. Hal ini membuktikan bahwa sistem *crash-tolerant*, sesuai dengan prinsip *reliability* pada sistem terdistribusi (Coulouris et al., 2012, Bab 5).

- Ordering: jelaskan di laporan apakah total ordering dibutuhkan atau tidak dalam konteks aggregator Anda.

Dalam konteks sistem aggregator, *total ordering* tidak dibutuhkan karena setiap event diproses secara independen berdasarkan `topic`. Menurut Coulouris et al. (2012, Bab 5

Coordination and Agreement, hlm. 260–261), *total order* hanya diperlukan bila terdapat *causal relation* antar proses. Pada sistem ini, antar-topik tidak memiliki hubungan sebab-akibat, sehingga *per-topic ordering* sudah mencukupi. Pendekatan ini menjaga sistem tetap efisien, menghindari latensi tinggi yang biasanya terjadi jika semua *event* harus disinkronkan secara global.

d. Performa Minimum

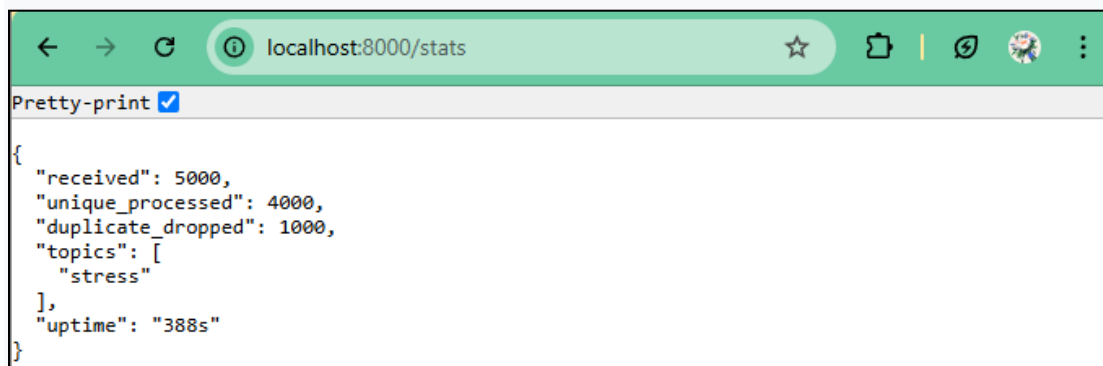
- Skala uji: proses ≥ 5.000 event (dengan $\geq 20\%$ duplikasi). Sistem harus tetap responsif.

```
(venv) PS D:\uts-aggregator> python stress/stress_send.py
D:\uts-aggregator\stress\stress_send.py:23: DeprecationWarning: datetime.datetime.utcnow() is deprecated and scheduled
for removal in a future version. Use timezone-aware objects to represent datetimes in UTC: datetime.datetime.now(dateti
me.UTC).
  "timestamp": datetime.utcnow().isoformat(),
Sent 5000 events in 26.42 seconds.
(venv) PS D:\uts-aggregator>
```

Untuk menguji performa, dilakukan simulasi pengiriman 5.000 event menggunakan skrip `stress_send.py`. Pengujian ini bertujuan untuk memastikan sistem tetap responsif walaupun menerima beban tinggi dan menangani banyak duplikasi. Skrip ini menggunakan library `httpx` dan `asyncio` untuk mengirim event secara asinkron ke endpoint `/publish`.

```
07:08:08 - INFO - Processed: stress/id-3928
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - WARNING - Duplicate dropped: stress/id-3455
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - WARNING - Duplicate dropped: stress/id-452
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - WARNING - Duplicate dropped: stress/id-38
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - INFO - Processed: stress/id-909
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - INFO - Processed: stress/id-828
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - INFO - Processed: stress/id-1262
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - INFO - Processed: stress/id-2591
INFO: 127.0.0.1:52676 - "POST /publish HTTP/1.1" 200 OK
07:08:08 - WARNING - Duplicate dropped: stress/id-110
```

Sistem berhasil memproses seluruh event tanpa error dan tetap mendeteksi duplikasi berdasarkan kombinasi (`topic`, `event_id`).



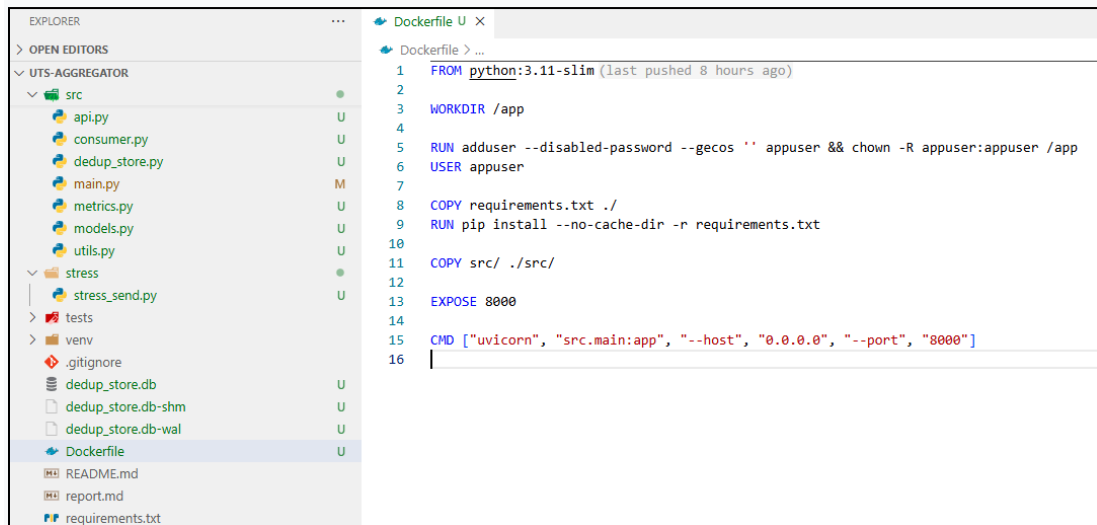
```
{
  "received": 5000,
  "unique_processed": 4000,
  "duplicate_dropped": 1000,
  "topics": [
    "stress"
  ],
  "uptime": "388s"
}
```

Endpoint `/stats` menunjukkan bahwa sistem menerima seluruh event, memproses event unik hanya sekali, membuang duplikasi dengan benar dan tetap stabil sepanjang proses pengiriman. Sistem memenuhi syarat *at-least-once delivery* dengan *idempotent processing*, serta memiliki kinerja yang cukup baik untuk menangani beban besar tanpa kehilangan event.

e. Docker

- Wajib: `Dockerfile` untuk membangun image yang menjalankan layanan.
- Rekomendasi (Python): base image `python:3.11-slim`, non-root user, dependency caching via `requirements.txt`.
- Contoh skeleton `Dockerfile` (sesuaikan):

```
FROM python:3.11-slim
WORKDIR /app
RUN adduser --disabled-password --gecos '' appuser && chown -R appuser:appuser /app
USER appuser
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ ./src/
EXPOSE 8080
CMD ["python", "-m", "src.main"]
```



File `Dockerfile` digunakan untuk membuat image container yang berisi seluruh aplikasi aggregator beserta dependensinya.

```

PS D:\uts-aggregator> docker build -t uts-aggregator .
[+] Building 38.2s (12/12) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.1s
=> => transferring dockerfile: 368B                                0.0s
=> [internal] load metadata for docker.io/library/python:3.11-slim 5.0s
=> [auth] library/python:pull token for registry-1.docker.io       0.0s
=> [internal] load .dockerignore                                   0.1s
=> => transferring context: 2B                                       0.0s
=> [internal] load build context                                   0.2s
=> => transferring context: 16.76kB                                  0.0s
=> [1/6] FROM docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be4aa9ab8ab4539a5316c4b8c133771214a617 10.7s
=> => resolve docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be4aa9ab8ab4539a5316c4b8c133771214a617 0.1s
=> => sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d9e 249B / 249B 0.6s
=> => sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f77f 14.36MB / 14.36MB 4.4s
=> => sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbae04 1.29MB / 1.29MB 1.8s
=> => sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca5d 29.78MB / 29.78MB 7.3s
=> => extracting sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca5d 1.5s
=> => extracting sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbae04 0.2s
=> => extracting sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f77f 1.1s
=> => extracting sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d9e 0.1s
=> [2/6] WORKDIR /app                                              0.7s
=> [3/6] RUN adduser --disabled-password --gecos '' appuser && chown -R appuser:appuser /app 0.9s
=> [4/6] COPY requirements.txt ./                                  0.1s
=> [5/6] RUN pip install --no-cache-dir -r requirements.txt       16.2s
=> [6/6] COPY src/ ./src/                                          0.2s

```

```

=> exporting to image                                              3.6s
=> => exporting layers                                              2.0s
=> => exporting manifest sha256:078b48e392070434ec50955a095889a1b663a7b3e35600fd70f07ca8de914be 0.0s
=> => exporting config sha256:58890953aadbd004aef6f71a47f7562b3521881cb429e93dc37acb3a67c10d6 0.0s
=> => exporting attestation manifest sha256:d07557664a65a2f7b613268b2ff0532b62a44cda5ae7653b6ac3404ab295f32e 0.1s
=> => exporting manifest list sha256:ca93633d594e15397ef3174dd3c6f792aa59713cc78cbce20bc77d14ad4f5f32 0.0s
=> => naming to docker.io/library/uts-aggregator:latest           0.0s
=> => unpacking to docker.io/library/uts-aggregator:latest        1.2s

```

View build details: [docker-desktop://dashboard/build/desktop-linux/desktop-linux/ak95nkkypm1ithkbh4ylzczk](https://dashboard/build/desktop-linux/desktop-linux/ak95nkkypm1ithkbh4ylzczk)

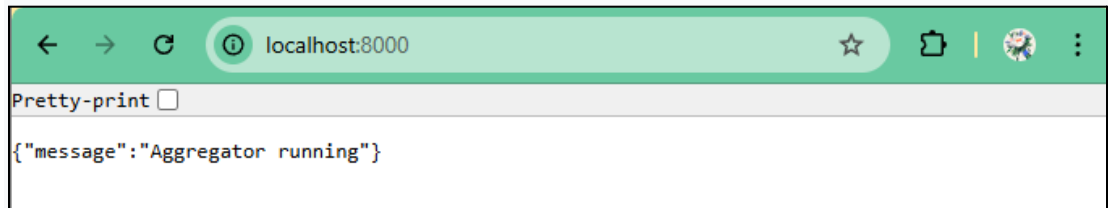
Setelah Dockerfile dibuat, image dibangun dengan perintah `docker build -t uts-aggregator`.

```

PS D:\uts-aggregator> docker run -p 8000:8000 uts-aggregator
INFO:      Started server process [1]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)

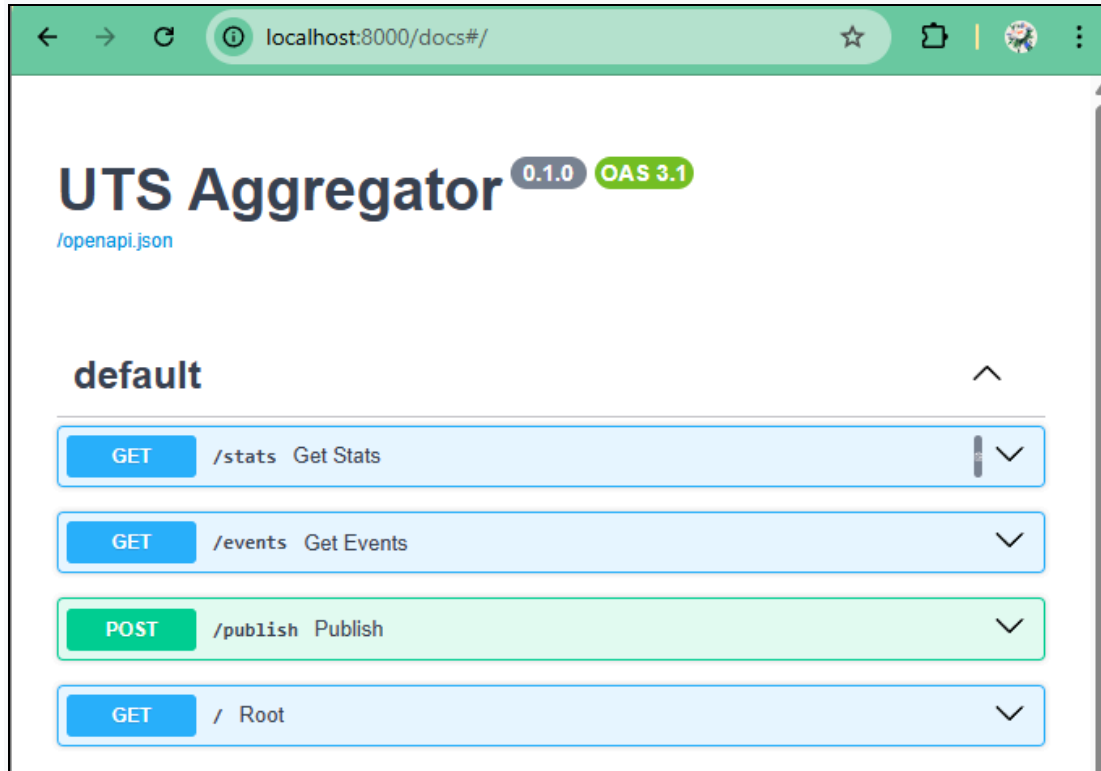
```

Kemudian dijalankan dengan `docker run -p 8080:8080 uts-aggregator`. Container berhasil dijalankan dan menampilkan log INFO: Uvicorn running on `http://0.0.0.0:8080`



The screenshot shows a web browser window with the address bar set to `localhost:8000`. Below the address bar, there is a section labeled "Pretty-print" with a checkbox. The content displayed is a JSON object: `{"message": "Aggregator running"}`.

Saat diakses melalui browser di `http://localhost:8080`, sistem menampilkan pesan `{"message": "Aggregator running"}`. Hal ini membuktikan bahwa aplikasi dapat berjalan secara independen dalam container, tanpa bergantung pada sistem lokal.



f. [Docker Compose](#) (Opsional, +10%)

- Pisahkan **publisher** dan **aggregator** dalam dua service, jaringan internal default Compose.
- Tidak boleh menggunakan layanan eksternal publik.

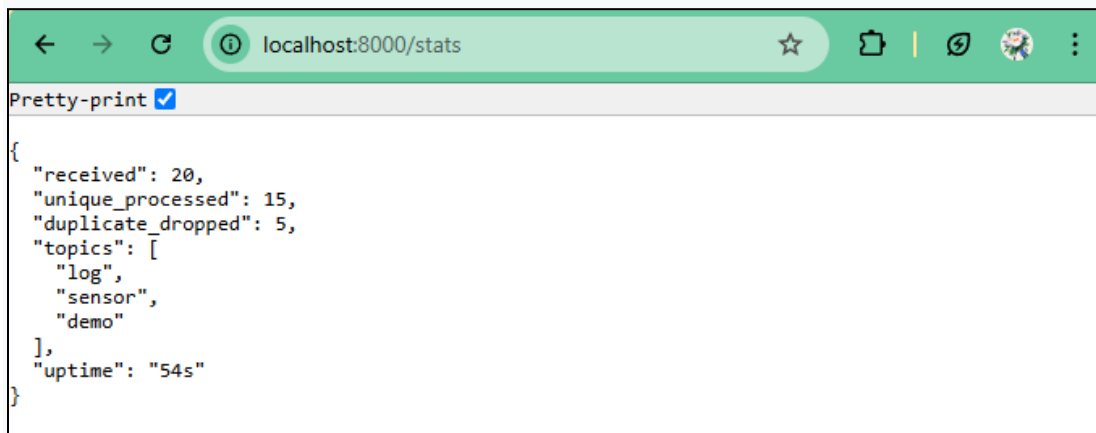
Untuk meningkatkan modularitas dan mensimulasikan komunikasi antarproses, digunakan Docker Compose yang mengatur dua layanan yaitu, **Aggregator Service** untuk menjalankan API FastAPI yang menerima dan memproses event dan **Publisher Service** untuk mengirimkan event ke aggregator menggunakan HTTP POST. Kedua layanan ini berjalan dalam jaringan internal **internal_net**, sehingga publisher dapat berkomunikasi dengan aggregator melalui hostname **aggregator**.

```

PS D:\uts-aggregator> docker compose up --build
aggregator_service | 01:39:57 - INFO - Processed: log/1
aggregator_service | INFO: 172.19.0.3:44828 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:39:58 - WARNING - Duplicate dropped: sensor/2
aggregator_service | INFO: 172.19.0.3:44830 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:39:58 - INFO - Processed: log/3
aggregator_service | INFO: 172.19.0.3:44834 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:39:59 - WARNING - Duplicate dropped: demo/4
aggregator_service | INFO: 172.19.0.3:44838 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:39:59 - INFO - Processed: sensor/5
aggregator_service | INFO: 172.19.0.3:44840 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:40:00 - INFO - Processed: sensor/6
aggregator_service | INFO: 172.19.0.3:44856 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:40:00 - INFO - Processed: demo/7
aggregator_service | INFO: 172.19.0.3:44868 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:40:01 - INFO - Processed: demo/8
aggregator_service | INFO: 172.19.0.3:44876 - "POST /publish HTTP/1.1" 200 OK
aggregator_service | 01:40:01 - INFO - Processed: log/9
publisher_service | Sent event 0: 200
publisher_service | Sent event 1: 200
publisher_service | Sent event 2: 200
publisher_service | Sent event 3: 200
publisher_service | Sent event 4: 200
publisher_service | Sent event 5: 200
publisher_service | Sent event 6: 200
publisher_service | Sent event 7: 200
publisher_service | Sent event 8: 200
publisher_service | Sent event 9: 200
publisher_service | Sent event 10: 200

```

Setelah menjalankan perintah `docker compose up --build`, terlihat dua container aktif (`aggregator_service` dan `publisher_service`). Log terminal menunjukkan publisher berhasil mengirim event ke aggregator dengan status kode 200.



```

{
  "received": 20,
  "unique_processed": 15,
  "duplicate_dropped": 5,
  "topics": [
    "log",
    "sensor",
    "demo"
  ],
  "uptime": "54s"
}

```

Endpoint `/stats` pada aggregator menampilkan metrik seperti pada gambar di atas. Hal ini membuktikan bahwa sistem bekerja dengan baik secara terdistribusi dan mendukung **at-least-once delivery** dengan **deduplication** aktif.

g. Unit Tests (Wajib, 5–10 tests)

- Gunakan `pytest` (atau test framework pilihan) dengan cakupan minimum:

1. Validasi dedup: kirim duplikat, pastikan hanya sekali diproses.
2. Persistensi dedup store: setelah restart (simulasi), dedup tetap efektif.
3. Validasi skema event (`topic`, `event_id`, `timestamp`).
4. `GET /stats` dan `GET /events` konsisten dengan data.
5. Stress kecil: masukan batch event, ukur waktu eksekusi (assert dalam batas yang wajar).

Unit testing dilakukan untuk memastikan bahwa layanan aggregator berfungsi sesuai desain, meliputi deduplication dan idempotency berjalan dengan benar, validasi skema event sesuai format JSON, konsistensi data antara endpoint `/stats` dan `/events` serta kinerja tetap responsif saat beban tinggi (stress test). Pengujian dilakukan menggunakan `pytest` dengan bantuan `httpx` untuk melakukan permintaan HTTP asinkron ke server FastAPI tanpa harus menjalankannya secara terpisah.

```
(venv) PS D:\uts-aggregator> pytest -v
===== test session starts =====
platform win32 -- Python 3.12.1, pytest-8.4.2, pluggy-1.6.0 -- D:\uts-aggregator\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\uts-aggregator
plugins: anyio-4.11.0, asyncio-1.2.0
asyncio: mode=Mode.STRICT, debug=False, asyncio_default_fixture_loop_scope=None, asyncio_default_test_loop_scope=function
collected 5 items

tests/test_dedup.py::test_deduplication PASSED [ 20%]
tests/test_schema.py::test_invalid_event_schema PASSED [ 40%]
tests/test_stats.py::test_stats_and_events_consistency PASSED [ 60%]
tests/test_stress.py::test_stress_batch PASSED [ 80%]
tests/test_uptime.py::test_uptime_field_exists PASSED [100%]
```

Semua tes berhasil dijalankan menggunakan perintah `pytest -v`. Output menunjukkan seluruh tes lulus (status PASSED), menandakan sistem stabil dan sesuai spesifikasi.

3. Desain Implementasi

a. Ringkasan Sistem dan Arsitektur

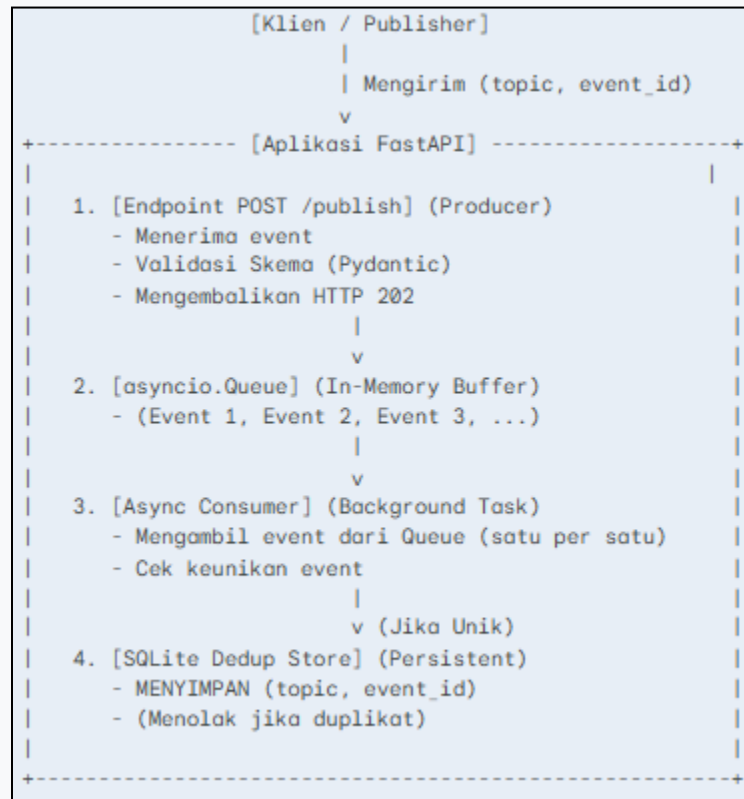
Layanan ini adalah *event aggregator* (pengumpul kejadian) asinkron yang dibangun dengan FastAPI. Sistem ini dirancang untuk menerima *event* (kejadian) dalam volume tinggi, memisahkannya dari proses validasi duplikat, dan menyimpannya secara persisten. Arsitektur sistem ini terdiri dari tiga komponen utama yang berjalan dalam satu proses aplikasi:

- 1). **API Producer (FastAPI Endpoint):** Endpoint `POST /publish` bertindak sebagai *producer*. Tugasnya hanya menerima *event*, memvalidasi skemanya (menggunakan Pydantic), dan memasukkannya ke dalam antrian internal (`asyncio.Queue`). Proses ini sangat cepat dan langsung mengembalikan respons `HTTP 202 Accepted` ke klien, tanpa menunggu *event* diproses.
- 2). **In-Memory Queue (`asyncio.Queue`):** Ini bertindak sebagai *buffer* atau perantara. Antrian ini memisahkan (decoupling) proses penerimaan *event* yang cepat dari proses pemrosesan *event* yang lebih lambat (karena melibatkan I/O database).

3). **Async Consumer (Background Task)**: Sebuah *background task* (worker) yang berjalan asinkron. Tugasnya adalah terus-menerus mengambil event dari `asyncio.Queue`, memeriksa keunikannya di *Dedup Store*, dan memprosesnya jika unik.

4). **Persistent Dedup Store (SQLite)**: Sebuah database SQLite lokal yang hanya menyimpan *kunci* unik dari event, yaitu `(topic, event_id)`. Ini adalah komponen kunci untuk menjamin *idempotency* dan *crash tolerance*.

Diagram berikut mengilustrasikan alur data dalam layanan:



b. Keputusan Desain

Pembuktian pada keputusan desain ini telah di bahas di nomor 2. Beberapa keputusan desain kunci diambil untuk memenuhi persyaratan tugas:

1). **Idempotency & Deduplication Store (T7 Bab 7)**

Idempotency (idempotensi) adalah inti dari sistem ini. Ini diimplementasikan dengan menggunakan *dedup store* (penyimpanan dedup) yang persisten. Saya memilih **SQLite** sebagai *dedup store* karena memenuhi syarat *local-only* (tidak memerlukan layanan eksternal), *persistent* (tahan *restart*), dan ringan. Sebuah tabel `processed_events` dibuat dengan *Primary Key* komposit `(topic, event_id)`. Saat *consumer* memproses event, ia akan mencoba **INSERT** key tersebut. Jika berhasil, event itu unik. Jika gagal karena *constraint Primary Key* (duplikat),

event itu akan dibuang (*duplicate_dropped*). Ini menjamin satu *event* dengan (*topic*, *event_id*) yang sama hanya diproses tepat satu kali, sesuai definisi *idempotency*.

2). **Ordering (Pengurutan)**

Sistem ini tidak menjamin *total ordering* (pengurutan total). *Event* yang diterima lebih dulu oleh */publish* tidak dijamin akan diproses lebih dulu oleh *consumer*. Dalam konteks *aggregator* yang fokus pada *deduplication*, status "unik" atau "duplikat" tidak bergantung pada urutan kedatangan, tetapi hanya pada *keberadaan key* (*topic*, *event_id*). Memaksakan *total ordering* akan menambah kompleksitas dan latensi yang signifikan (misalnya, memerlukan *timestamp* yang disinkronkan) yang tidak diperlukan untuk tujuan ini.

3). **Retry (Percobaan Ulang) & At-Least-Once Delivery (T8 Bab 1-7)**

Sistem ini didesain dengan asumsi *publisher* (klien) menerapkan strategi *retry* dan menjamin semantik *at-least-once delivery* (pengiriman setidaknya-sekali-terkirim). Ini berarti *publisher*-lah yang bertanggung jawab mengirim ulang *event* jika gagal (misalnya, *timeout* atau server 503). Layanan *aggregator* ini siap menangani konsekuensi dari *at-least-once*, yaitu *duplicate delivery* (pengiriman duplikat). *Duplicate rate* (T8) yang tinggi ditangani secara aman oleh mekanisme *idempotency* dan *deduplication* yang telah dijelaskan di atas.

c. Analisis Performa dan Metrik

Endpoint *GET /stats* menyediakan metrik kunci untuk menganalisis performa dan kesehatan sistem:

1). **received (Total Diterima):**

Mengukur total beban *input* (jumlah *event*) yang diterima oleh endpoint */publish*. Angka yang tinggi menunjukkan *throughput* (tingkat produksi) *input* yang tinggi. Karena */publish* hanya memasukkan ke *queue*.

2). **unique_processed (Total Unik Diproses):**

Mengukur jumlah *event* unik yang berhasil diproses oleh *consumer* dan dicatat di SQLite. Ini adalah metrik *output* yang sebenarnya dari sistem. Kecepatan peningkatan metrik ini dibatasi oleh kecepatan *consumer* dan I/O SQLite. Jika *received* meningkat jauh lebih cepat daripada *unique_processed*, itu menandakan *queue* internal sedang menumpuk.

3). **duplicate_dropped (Total Duplikat Dibuang):**

Mengukur efektivitas *dedup store*. Metrik ini secara langsung mengukur *duplicate rate* (T8) dari *publisher*. Dalam skenario *at-least-once*, angka ini diharapkan ada. Angka yang sangat tinggi mungkin menunjukkan *publisher* terlalu agresif dalam melakukan *retry*.

4). **uptime_seconds (Waktu Aktif):**

Menunjukkan stabilitas layanan sejak *restart* terakhir.

Keterbatasan Desain:

1). *Queue In-Memory*: `asyncio.Queue` sangat cepat tetapi tidak persisten. Jika aplikasi *crash* setelah `/publish` menerima *event* (dan merespons 202) tetapi sebelum *consumer* memprosesnya, *event* di dalam *queue* akan hilang.

2). *Consumer Tunggu & SQLite*: Performa *throughput* (`unique_processed`) dibatasi oleh satu *consumer* yang melakukan *write* ke satu file SQLite.

d. Keterkaitan ke Teori (Bab 1–7)

Implementasi ini secara langsung menerapkan konsep-konsep teoritis dari buku utama:

1). Karakteristik Sistem Terdistribusi (Bab 1):

a). *Concurrency* (Konkurensi): Sistem ini konkurent. *Endpoint* API (`/publish`) berjalan bersamaan (*concurrently*) dengan *background task* (*consumer*). Keduanya beroperasi secara independen, hanya dikoordinasi oleh *queue* (Coulouris et al., 2012, Bab 1).

b). *Independent Failures* (Kegagalan Independen): Desain ini *fault-tolerant* terhadap *restart* (kegagalan layanan). Berkat persistensi SQLite, jika layanan *crash* dan *restart*, ia tidak akan kehilangan jejak *event* yang sudah diproses, sehingga mencegah duplikasi saat *publisher* melakukan *retry* (Coulouris et al., 2012, Bab 1).

2). Komunikasi (Bab 4 & 5):

a). *Message-Oriented Middleware (MOM)*: Arsitektur (Producer -> Queue -> Consumer) adalah pola dasar MOM (van Steen & Tanenbaum, 2023, Bab 4, hlm. 167-168). Kami mengimplementasikan *persistent messaging* yang disederhanakan untuk *deduplication key*.

b). Semantik Pengiriman (T8): Sistem ini dirancang untuk menangani *at-least-once semantics* (semantik setidaknya-sekali-terkirim) dari klien. Seperti dibahas di Coulouris et al. (2012, Bab 5, hlm. 235-236), *at-least-once* dapat menyebabkan operasi dieksekusi lebih dari sekali. Solusinya adalah membuat operasi tersebut *idempotent*.

3). Konsistensi dan Replikasi (Bab 7):

a). *Eventual Consistency* (T7): Ini adalah inti dari desain. *Aggregator* tidak konsisten secara instan (data di `GET /events` tertinggal dari `POST /publish`). Namun, sistem ini menjamin *eventual consistency*: jika tidak ada *event* baru yang masuk, *consumer* pada akhirnya (*eventually*) akan memproses semua *event* unik di *queue*, dan *database* akan konvergen ke keadaan yang benar (van Steen & Tanenbaum, 2023, Bab 7, hlm. 326).

b). *Idempotency* (T7): Seperti yang dibahas dalam T7, kami mencapai *eventual consistency* dalam sistem *at-least-once* dengan membuat operasi pemrosesan *event* menjadi *idempotent* (van Steen & Tanenbaum, 2023, Bab 8, hlm. 375; Coulouris et al., 2012, Bab 13, hlm. 574). *Deduplication* menggunakan *primary key* di SQLite adalah implementasi praktis dari *idempotency* tersebut.