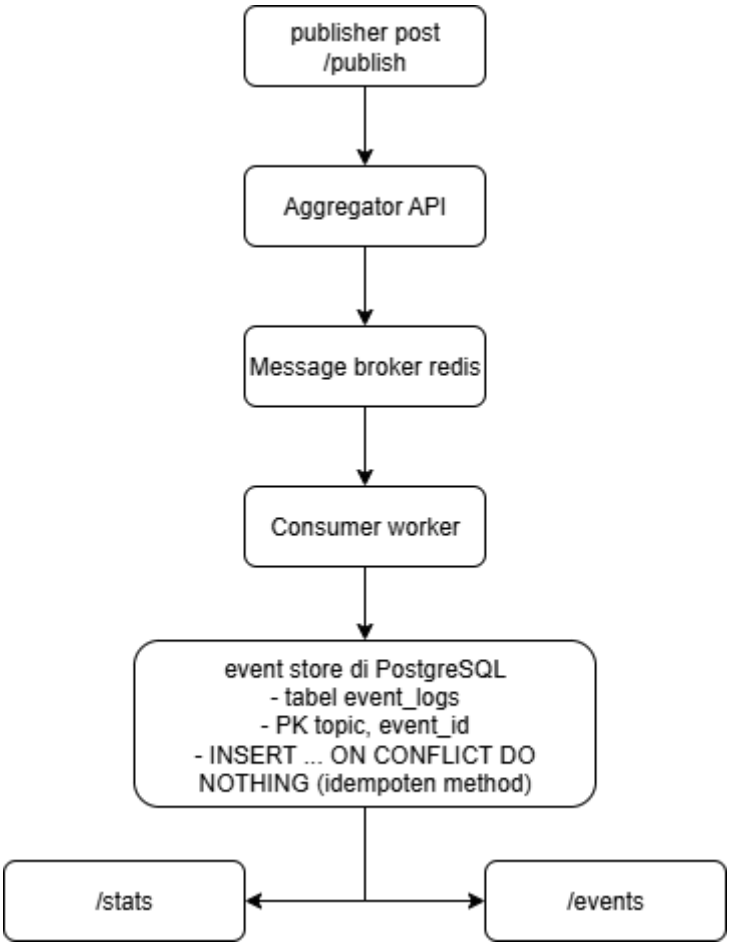
Nama : Erizki Fadli
NIM : 11221014
Kelas : SisTer B

UAS

Ringkasan sistem dan arsitektur



1. Publisher: berfungsi untuk produsen data event. Komponen ini berjalan pada container terpisah dan akan terus mengirim event kepada API aggregator. Pada konfigurasi saat ini, container hanya akan berjalan untuk mengirim 20000 event dengan 30% diantaranya adalah data duplikat.
2. Aggregator API: Berfungsi untuk menerima request event dan memvalidasi request menggunakan pydantic. Setelah berhasil divalidasi, data diteruskan ke antrean Redis.
3. Message broker Redis: berfungsi sebagai penampung seluruh event yang masuk sebelum diproses oleh worker.
4. Consumer worker: Berfungsi untuk mengambil data dari antrean redis dan memproses untuk ditulis ke db.
5. Event store PostgreSQL: Berfungsi sebagai penyimpanan persisten sekaligus validasi race condition menggunakan perintah sql on conflict do nothing. Jika dua worker

memasukkan id bersamaan, db akan mengunci event tersebut dan mengabaikan yang lainnya tanpa error.

6. /stats dan /events: berfungsi untuk endpoint statistik diantranya total_received, unique_processed, dan lain-lain.

Keputusan desain

1. Idempotency pada consumer
   Pola Idempotent Consumer dipilih sebagai strategi utama untuk menangani jaminan pengiriman at-least-once dari Message Broker. Dengan memanfaatkan Composite Primary Key (kombinasi topic dan event_id), sistem menjamin bahwa satu event unik hanya akan memiliki satu state akhir di database. Meskipun pesan yang sama dikirimkan berkali-kali oleh publisher atau dikirim ulang oleh broker, hasil pemrosesan data tetap tunggal dan konsisten.
2. Dedup store pada PostgreSQL
   Sistem menggunakan PostgreSQL sebagai deduplication store persisten dengan mekanisme unique constraint pada kolom (topic, event_id). PostgreSQL dipilih karena kemampuan konkurensi yang lebih tinggi dan dukungan row-level locking dengan mekanisme penyimpanan dilakukan pada docker volume (pg_data) sehingga status deduplikasi tetap terjaga meskipun container mengalami crash atau restart.
3. Total Ordering
   Sistem tidak mengimplementasikan total ordering dikarenakan perannya sudah diambil alih oleh properti processed_at dan timestamp yang tercantum pada setiap data publish yang masuk dan properti ini dapat digunakan untuk mengurutkan data dengan konsisten tanpa mengorbankan performa.
4. Retry
   Untuk mencegah masalah retry, sistem mengimplementasikan fail-safe dimana jika consumer mengalami crash saat memproses pesan tapi belum melakukan ack ke broker, maka broker akan mengirim ulang pesan. Mekanisme ini dijelaskan pada poin 1.
5. Race condition
   Untuk mencegah race condition dimana beberapa worker memproses event id yang sama disaat bersamaan, sistem menerapkan atomic database transaction dimana sistem tidak menggunakan pengecekan di level aplikasi melainkan sistem menyerahkan kontrol konkurensi sepenuhnya ke DB menggunakan perintah sql. Dengan ini DB akan mengunci index unik saat transaksi terjadi dan jika dua worker masuk bersamaan, satu akan diproses dan satu lagi akan gagal karena sudah ada index sebelumnya yang sama.

Analisis Performa dan Metrik

Pada program terdapat 12 unit test yang menguji fungsionalitas dan performa sistem yakni dedup, persistence, schema, get stats dan events, dan stress test dengan 20000 data dan 30% duplikat dengan hasil sebagai berikut:

1. Single worker integration test

```
PS C:\Users\Fadli\Documents\KULIAH\E LEARNING\sms 7\sister\uas\uas_aggregator> docker compose exec aggregator pytest /tests/test_integration.py -v
============================= test session starts ==============================
platform linux -- Python 3.11.14, pytest-8.0.0, pluggy-1.6.0 -- /usr/local/bin/python3.11
cachedir: .pytest_cache
rootdir: /tests
plugins: anyio-4.12.0, asyncio-0.23.5
asyncio: mode=Mode.STRICT
collected 12 items

../tests/test_integration.py::test_1_health_check PASSED                 [  8%]
../tests/test_integration.py::test_2_publish_valid_event PASSED          [ 16%]
../tests/test_integration.py::test_3_deduplication_logic PASSED          [ 25%]
../tests/test_integration.py::test_4_stats_consistency PASSED            [ 33%]
../tests/test_integration.py::test_5_invalid_schema PASSED               [ 41%]
../tests/test_integration.py::test_6_persistence_check PASSED            [ 50%]
../tests/test_integration.py::test_7_get_events_filter PASSED            [ 58%]
../tests/test_integration.py::test_8_concurrency_simulation PASSED       [ 66%]
../tests/test_integration.py::test_9_batch_publish PASSED                [ 75%]
../tests/test_integration.py::test_10_database_connection PASSED         [ 83%]
../tests/test_integration.py::test_11_timestamp_parsing PASSED           [ 91%]
../tests/test_integration.py::test_12_payload_integrity PASSED           [100%]

============================== warnings summary ===============================
test_integration.py::test_1_health_check
  /usr/local/lib/python3.11/site-packages/pytest_asyncio/plugin.py:769: DeprecationWarning: The event_loop fixture provided by pytest-asyncio has been redefined in
  /tests/test_integration.py:11
  Replacing the event_loop fixture with a custom implementation is deprecated
  and will lead to errors in the future.
  If you want to request an asyncio event loop with a scope other than function
  scope, use the "scope" argument to the asyncio mark when marking the tests.
  If you want to return different types of event loops, use the event_loop_policy
  fixture.

    warnings.warn(

test_integration.py::test_12_payload_integrity
  /usr/local/lib/python3.11/site-packages/_pytest/fixtures.py:1064: RuntimeWarning: coroutine 'setup_db' was never awaited
    self.cached_result = None
  Enable tracemalloc to get traceback where the object was allocated.
  See https://docs.pytest.org/en/stable/how-to/capture-warnings.html#resource-warnings for more info.

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 12 passed, 2 warnings in 7.27s ========================
```

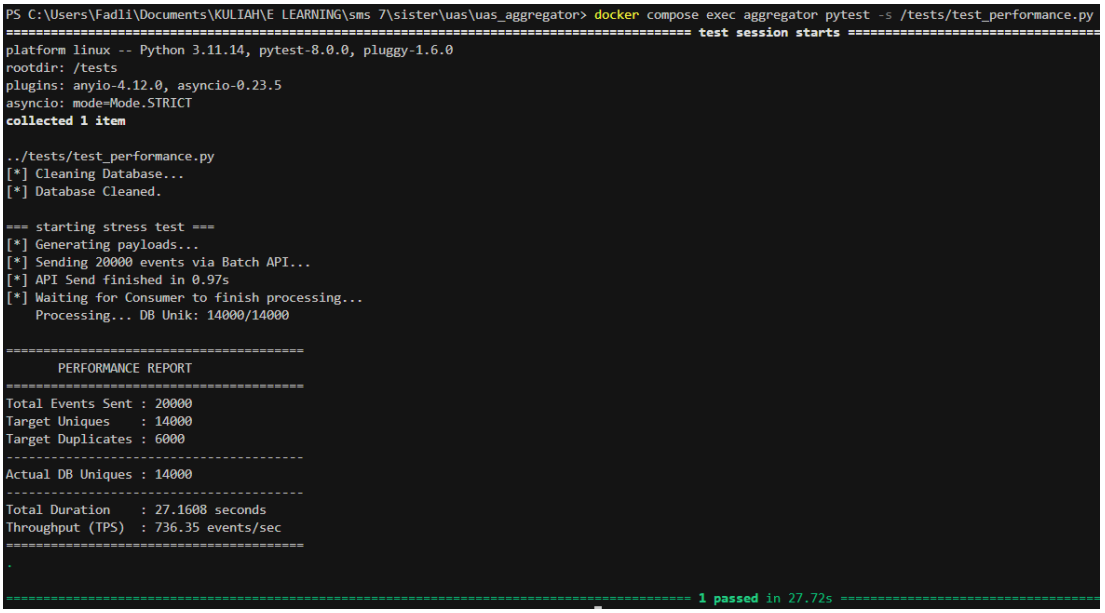
2. Single worker stress test

```
PS C:\Users\Fadli\Documents\KULIAH\E LEARNING\sms 7\sister\uas\uas_aggregator> docker compose exec aggregator pytest -s /tests/test_performance.py
============================= test session starts ==============================
platform linux -- Python 3.11.14, pytest-8.0.0, pluggy-1.6.0
rootdir: /tests
plugins: anyio-4.12.0, asyncio-0.23.5
asyncio: mode=Mode.STRICT
collected 1 item

../tests/test_performance.py
[*] Cleaning Database...
[*] Database Cleaned.

=== starting stress test ===
[*] Generating payloads...
[*] Sending 20000 events via Batch API...
[*] API Send finished in 0.75s
[*] Waiting for Consumer to finish processing...
    Processing... DB Unik: 14000/14000

========================================
       PERFORMANCE REPORT
========================================
Total Events Sent : 20000
Target Uniques    : 14000
Target Duplicates : 6000
----------------------------------------
Actual DB Uniques : 14000
----------------------------------------
Total Duration    : 72.1115 seconds
Throughput (TPS)  : 277.35 events/sec
========================================
.

============================== 1 passed in 72.70s (0:01:12) ==============================
```

3. 3 worker integration test

```
PS C:\Users\Fadli\Documents\KULIAH\E LEARNING\sms 7\sister\uas\uas_aggregator> docker compose exec aggregator pytest /tests/test_integration.py -v
============================= test session starts ==============================
platform linux -- Python 3.11.14, pytest-8.0.0, pluggy-1.6.0 -- /usr/local/bin/python3.11
cachedir: .pytest_cache
rootdir: /tests
plugins: anyio-4.12.0, asyncio-0.23.5
asyncio: mode=Mode.STRICT
collected 12 items

../tests/test_integration.py::test_1_health_check PASSED                 [  8%]
../tests/test_integration.py::test_2_publish_valid_event PASSED          [ 16%]
../tests/test_integration.py::test_3_deduplication_logic PASSED          [ 25%]
../tests/test_integration.py::test_4_stats_consistency PASSED            [ 33%]
../tests/test_integration.py::test_5_invalid_schema PASSED               [ 41%]
../tests/test_integration.py::test_6_persistence_check PASSED            [ 50%]
../tests/test_integration.py::test_7_get_events_filter PASSED            [ 58%]
../tests/test_integration.py::test_8_concurrency_simulation PASSED       [ 66%]
../tests/test_integration.py::test_9_batch_publish PASSED                [ 75%]
../tests/test_integration.py::test_10_database_connection PASSED         [ 83%]
../tests/test_integration.py::test_11_timestamp_parsing PASSED           [ 91%]
../tests/test_integration.py::test_12_payload_integrity PASSED           [100%]

============================== warnings summary ===============================
test_integration.py::test_1_health_check
  /usr/local/lib/python3.11/site-packages/pytest_asyncio/plugin.py:769: DeprecationWarning: The event_loop fixture provided by pytest-asyncio has been redefined in
  /tests/test_integration.py:11
  Replacing the event_loop fixture with a custom implementation is deprecated
  and will lead to errors in the future.
  If you want to request an asyncio event loop with a scope other than function
  scope, use the "scope" argument to the asyncio mark when marking the tests.
  If you want to return different types of event loops, use the event_loop_policy
  fixture.

    warnings.warn(

test_integration.py::test_12_payload_integrity
  /usr/local/lib/python3.11/site-packages/_pytest/fixtures.py:1064: RuntimeWarning: coroutine 'setup_db' was never awaited
    self.cached_result = None
  Enable tracemalloc to get traceback where the object was allocated.
  See https://docs.pytest.org/en/stable/how-to/capture-warnings.html#resource-warnings for more info.

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 12 passed, 2 warnings in 7.50s ========================
```

4. 3 worker stress test



Keterkaitan Sistem Dengan Bab 1-13 Buku Utama

T1: Karakteristik Sistem Terdistribusi & Trade-off

Sistem Pub-Sub log aggregator yang dibangun mencerminkan karakteristik utama sistem terdistribusi, yaitu pemisahan fungsional (functional separation) antara komponen pengirim (publisher), perantara (broker), dan pemroses (aggregator/worker). Desain ini mengadopsi prinsip distribution transparency, khususnya location transparency, di mana publisher hanya perlu mengetahui alamat API endpoint tanpa perlu mengetahui lokasi fisik worker atau database PostgreSQL. Sistem menerapkan middleware (Redis & FastAPI) untuk menyembunyikan heterogenitas platform dan memfasilitasi komunikasi asinkron. Trade-off utama dalam desain ini adalah mengorbankan strict performance (akibat overhead serialisasi JSON dan network hops ke Redis/DB) demi mendapatkan scalability dan availability. Sistem menerima bahwa kegagalan parsial (partial failure) seperti koneksi Redis putus adalah hal wajar, sehingga dirancang dengan mekanisme pemulihan otomatis via orkestrasi kontainer (van Steen & Tanenbaum, 2023).

T2 : Arsitektur Publish–Subscribe vs Client–Server

Arsitektur publish–subscribe dipilih menggantikan model client–server tradisional (request-response sinkron) karena kebutuhan akan decoupling dalam dimensi ruang dan waktu (space and time decoupling). Dalam sistem ini, publisher dan consumer tidak perlu aktif pada saat yang bersamaan; publisher dapat terus mengirim event ke Redis (broker) meskipun worker sedang restart atau sibuk (time decoupling). Redis bertindak sebagai Message-Oriented Middleware (MOM) yang memfasilitasi antrean persisten. Berbeda dengan arsitektur layered murni, model ini memungkinkan penambahan jumlah worker secara dinamis tanpa mengubah kode di sisi publisher (referential decoupling). Hal ini sesuai dengan teori arsitektur data-centered di mana repositori data (Postgres) dan antrean pesan

(Redis) menjadi pusat integrasi komponen yang terdistribusi longgar (van Steen & Tanenbaum, 2023).

T3: At-least-once Delivery & Peran Idempotent Consumer

Dalam komunikasi terdistribusi menggunakan protokol HTTP dan Redis, sistem menghadapi tantangan reliabilitas jaringan. Sistem mengadopsi semantik pengiriman at-least-once delivery, di mana pesan dijamin sampai minimal satu kali, namun duplikasi mungkin terjadi akibat mekanisme retry otomatis saat terjadi timeout atau kegagalan acknowledgment. Untuk menangani ini, peran idempotent consumer menjadi kritikal. Worker dirancang untuk memproses pesan dengan operasi yang aman (idempoten), memastikan bahwa pemrosesan pesan yang sama berulang kali (karena duplikasi) menghasilkan state akhir database yang sama persis seperti jika pesan hanya diproses sekali. Ini menjembatani kesenjangan antara realitas jaringan yang tidak sempurna (message duplication) dengan kebutuhan aplikasi akan akurasi data (exactly-once processing) (van Steen & Tanenbaum, 2023).

T4: Skema Penamaan (Naming) untuk Deduplikasi

Agar deduplikasi dapat berjalan efektif, sistem memerlukan skema penamaan yang bersifat location-independent dan globally unique. Sistem menggunakan kombinasi atribut topic (sebagai namespace logis) dan event_id (sebagai identifier unik, misal UUID) untuk membentuk Composite Key. Sesuai teori flat naming, event_id ini tidak memuat informasi lokasi atau rute, melainkan hanya identitas murni dari event tersebut. Kunci komposit (topic, event_id) ini dijadikan Unique Constraint pada tabel PostgreSQL. Dengan skema ini, resolusi nama (name resolution) untuk mendeteksi duplikasi dilakukan langsung oleh mesin database melalui indeks unik, yang jauh lebih efisien dan collision-resistant dibandingkan pengecekan manual di level aplikasi (van Steen & Tanenbaum, 2023).

T5: Ordering & Sinkronisasi Waktu

Sistem terdistribusi tidak memiliki jam global (global clock) yang sempurna. Oleh karena itu, sistem ini tidak memaksakan total ordering ketat berdasarkan waktu kedatangan (arrival time) di broker, karena network latency dapat menyebabkan pesan yang dikirim duluan sampai belakangan. Sebagai gantinya, sistem menggunakan ordering berbasis konten (content-based ordering) menggunakan timestamp yang dibangkitkan oleh publisher (sumber kebenaran logis). Meskipun physical clock antar node mungkin mengalami skew, penggunaan timestamp dari sumber tunggal (publisher) dikombinasikan dengan mekanisme penyimpanan PostgreSQL memungkinkan pengurutan data secara logis saat query (ORDER BY timestamp). Ini adalah pendekatan praktis untuk menghindari kompleksitas algoritma sinkronisasi jam logis seperti Lamport timestamps yang berlebihan untuk kasus log agregator sederhana (van Steen & Tanenbaum, 2023).

T6: Failure Modes & Mitigasi (Persistensi)

Sistem dirancang untuk tahan terhadap berbagai failure modes, termasuk crash failure pada worker dan omission failure pada jaringan. Mitigasi utama dilakukan melalui persistensi data

(data persistence). Penggunaan Docker Volumes (pg_data untuk Postgres dan broker_data untuk Redis) memastikan durabilitas data. Jika container worker atau database mengalami crash dan di-restart oleh Docker, data antrean di Redis (jika mode AOF/RDB aktif) dan data event di PostgreSQL tidak akan hilang. Selain itu, mekanisme acknowledgment manual (atau implicit via transaction commit) memastikan pesan tidak hilang dari antrean sebelum sukses disimpan ke disk. Ini sesuai dengan prinsip recovery dalam sistem terdistribusi, di mana state sistem dipulihkan dari penyimpanan stabil (stable storage) pasca-kegagalan (van Steen & Tanenbaum, 2023).

T7: Konsistensi Data (Eventual Consistency)

Dalam konteks agregator log, sistem menerapkan model data-centric consistency, spesifiknya yang mendekati sequential consistency dari sudut pandang penyimpanan. Ketika event masuk ke antrean, terdapat jeda waktu (latency) sebelum data tersedia untuk dibaca di API /events. Namun, berkat mekanisme deduplikasi yang kuat di PostgreSQL, sistem menjamin konvergensi data (state convergence). Semua replika pembaca (jika ada) atau request berulang ke API /stats pada akhirnya akan melihat jumlah data unik yang sama, terlepas dari berapa kali event duplikat dikirim. Idempotency dan deduplikasi bertindak sebagai mekanisme resolusi konflik otomatis, memastikan bahwa anomali duplikasi tidak merusak integritas data dalam jangka panjang (van Steen & Tanenbaum, 2023).

T8 (Bab 8): Desain Transaksi (ACID & Isolation)

Transaksi basis data adalah komponen kunci dalam menjaga integritas sistem. Sistem menggunakan PostgreSQL yang mendukung properti ACID (Atomicity, Consistency, Isolation, Durability). Operasi penyisipan data event diperlakukan sebagai satu unit atomik.

- Atomicity: Saat worker melakukan INSERT, operasi tersebut sukses sepenuhnya atau gagal sepenuhnya. Tidak ada kejadian "setengah data" tersimpan.
- Isolation: PostgreSQL menggunakan isolation level default READ COMMITTED. Ini mencegah dirty reads, memastikan endpoint /stats hanya menghitung data yang benar-benar telah di-commit. Strategi ini menghindari masalah lost-update tanpa perlu locking agresif yang menghambat performa, karena setiap event memiliki identitas unik yang immutable.

T9: Kontrol Konkurensi & Race Condition

Dalam lingkungan multi-worker (konkurensi tinggi), race condition terjadi ketika dua worker mencoba memproses event dengan ID yang sama secara bersamaan. Sistem menangani ini bukan dengan distributed locking (misal: Redis Lock) yang kompleks, melainkan mendelegasikannya ke Database Management System (DBMS) (van Steen & Tanenbaum, 2023). Mekanismenya adalah menggunakan fitur ON CONFLICT DO NOTHING pada PostgreSQL. Ini adalah bentuk pessimistic concurrency control di level baris data (row-level locking) pada indeks unik. Ketika Transaksi A sedang memasukkan event_id=X, Transaksi B yang mencoba memasukkan ID yang sama akan diblokir sementara hingga A selesai. Jika A

sukses commit, B akan mendeteksi konflik dan mengabaikan operasi tersebut (DO NOTHING). Ini menjamin bahwa dalam kondisi paralelisme tinggi sekalipun, konsistensi data tetap terjaga.

T10: Orkestrasi & Virtualisasi (Docker Compose)

Sistem memanfaatkan virtualisasi tingkat sistem operasi (containerization) melalui Docker. Docker Compose bertindak sebagai orkestrator lokal yang mendefinisikan topologi sistem. Setiap layanan (Aggregator, Publisher, Redis, Postgres) berjalan di ruang pengguna (user space) terisolasi, mencegah konflik dependensi. Koneksi antar container menggunakan private bridge network (internal_net), sehingga layanan database dan broker tidak terekspos ke jaringan publik, meningkatkan keamanan. Docker menyediakan abstraksi log terpusat yang memudahkan pemantauan aktivitas kontainer secara real-time. Pendekatan ini selaras dengan tren sistem terdistribusi modern yang menggeser kompleksitas manajemen proses ke infrastruktur orkestrasi.

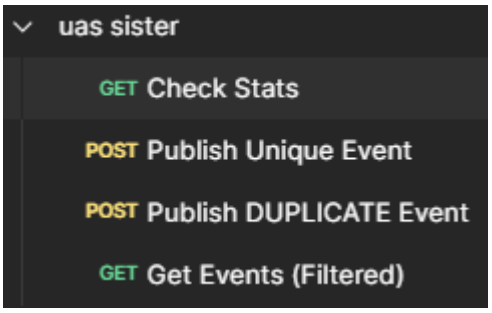
Bagian Implementasi

a. Arsitektur layanan

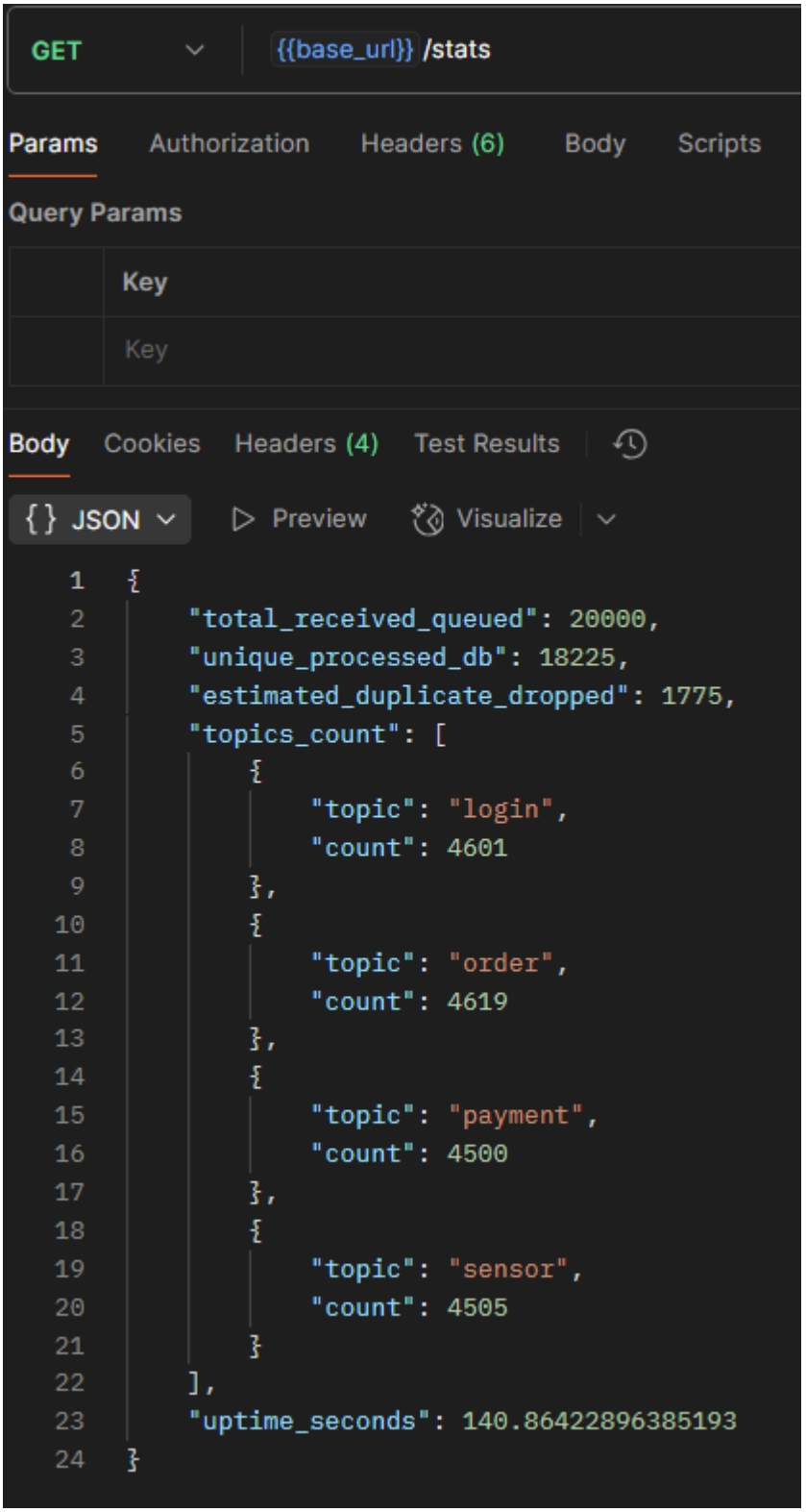
1. Publisher: berfungsi untuk produsen data event. Komponen ini berjalan pada container terpisah dan akan terus mengirim event kepada API aggregator. Pada konfigurasi saat ini, container hanya akan berjalan untuk mengirim 20000 event dengan 30% diantaranya adalah data duplikat.
2. Aggregator API: Berfungsi untuk menerima request event dan memvalidasi request menggunakan pydantic. Setelah berhasil divalidasi, data diteruskan ke antrean Redis.
3. Message broker Redis: berfungsi sebagai penampung seluruh event yang masuk sebelum diproses oleh worker.
4. Consumer worker: Berfungsi untuk mengambil data dari antrean redis dan memproses untuk ditulis ke db.
5. Event store PostgreSQL: Berfungsi sebagai penyimpanan persisten sekaligus validasi race condition menggunakan perintah sql on conflict do nothing. Jika dua worker memasukkan id bersamaan, db akan mengunci event tersebut dan mengabaikan yang lainnya tanpa error.
6. /stats dan /events: berfungsi untuk endpoint statistik diantranya total_received, unique_processed, dan lain-lain.
7. Network antar container sepenuhnya lokal.

b. Model event & API

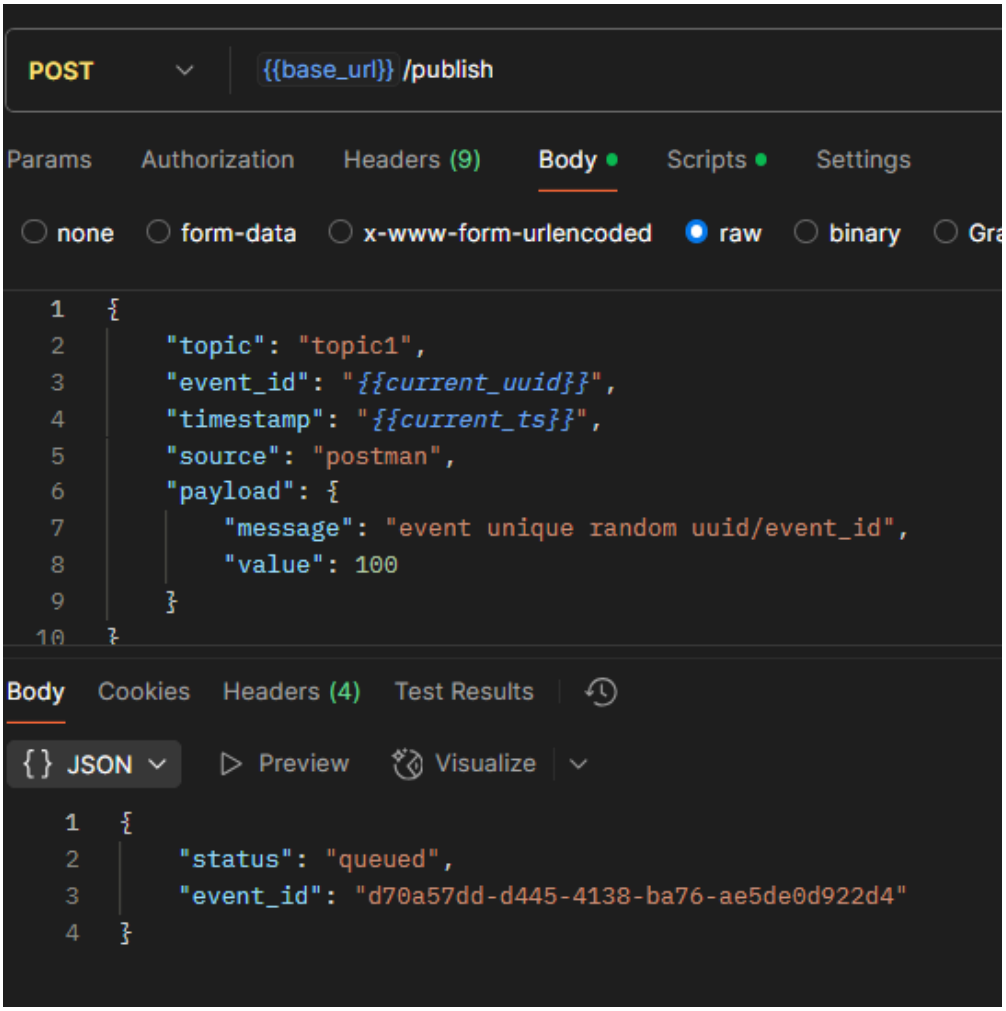
Sistem diimplementasikan dengan tiga endpoint berikut:



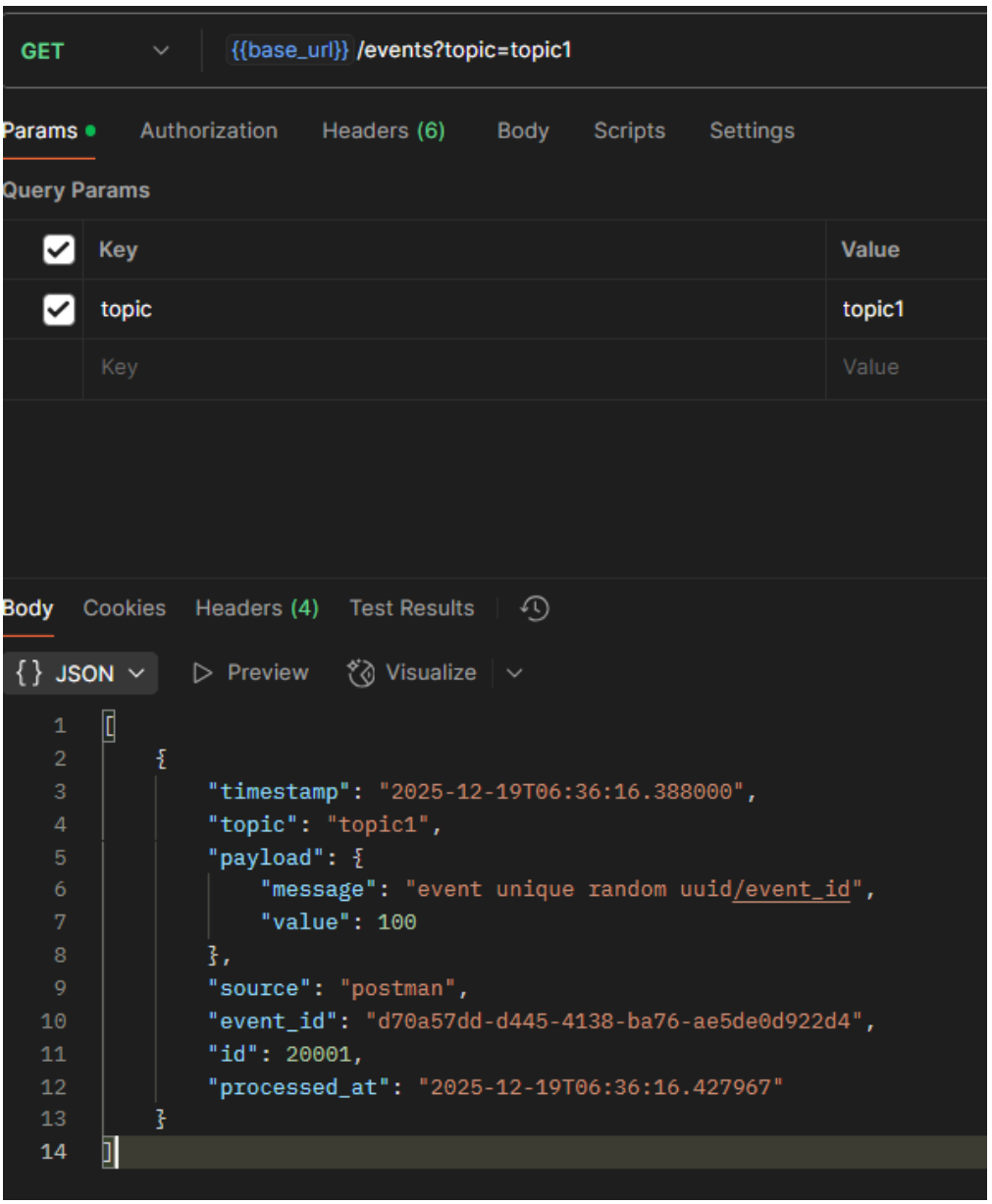
- Get /stats: berfungsi untuk melihat metrik sistem



- /publish: berfungsi untuk mengirimkan data untuk diantrekan/enqueu in memory.

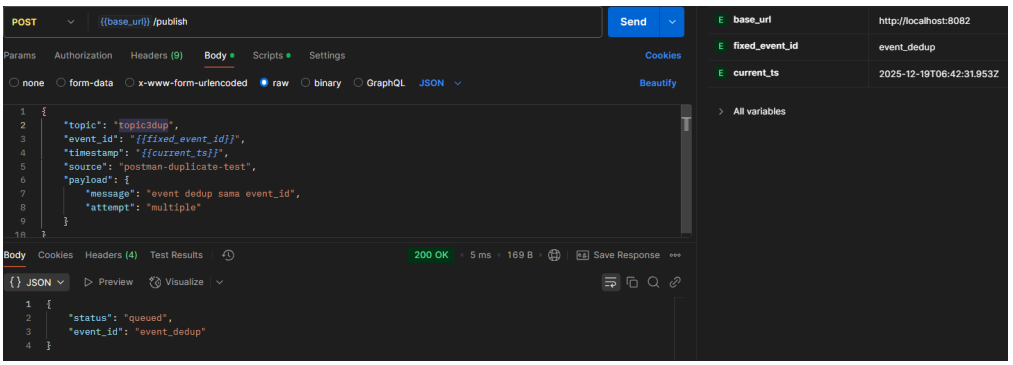


- /events: berfungsi untuk menampilkan seluruh event yang disimpan jika tanpa parameter topic, dan detail sebuah topic jika menggunakan parameter nama topic.

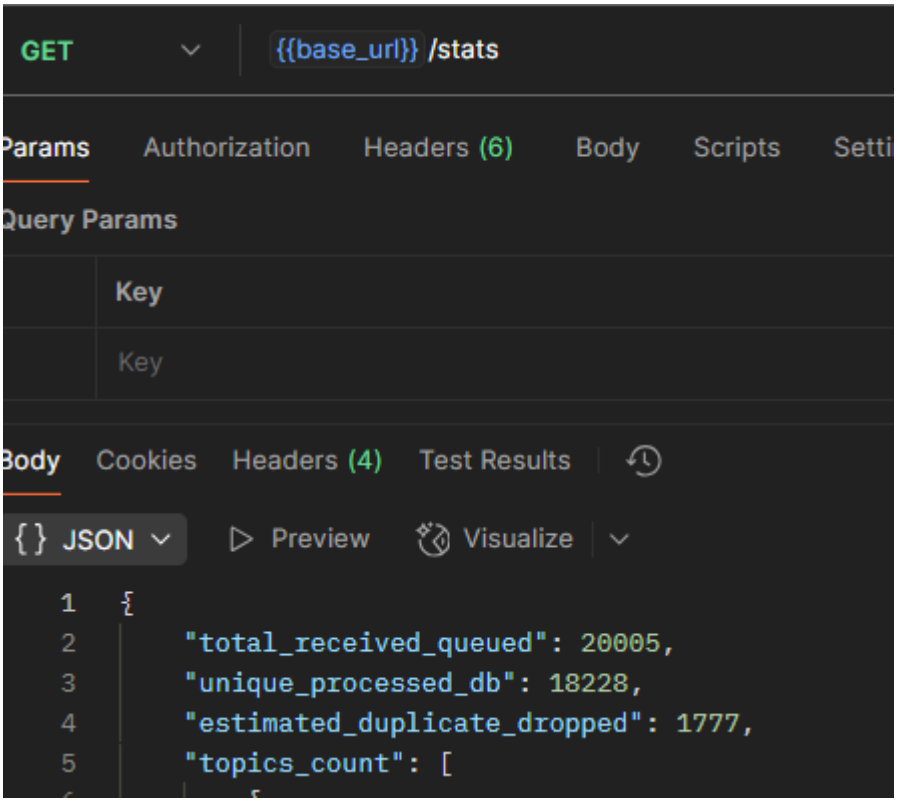


c. Idempotency & deduplication

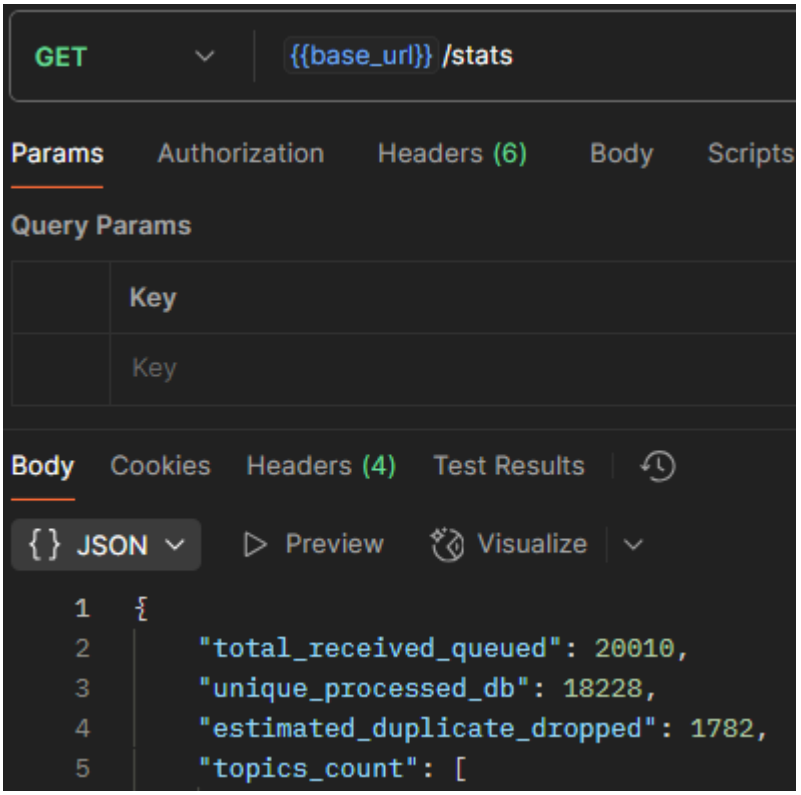
Dedup store menggunakan basis data PostgreSQL yang mana data tersimpan secara permanen sehingga tidak terpengaruh restart server. Idempotency diimplementasikan dengan pengecekan PK topic dan event_id. Jika ada yang keduanya sama persis dengan yang sudah disimpan, maka akan dibuang.



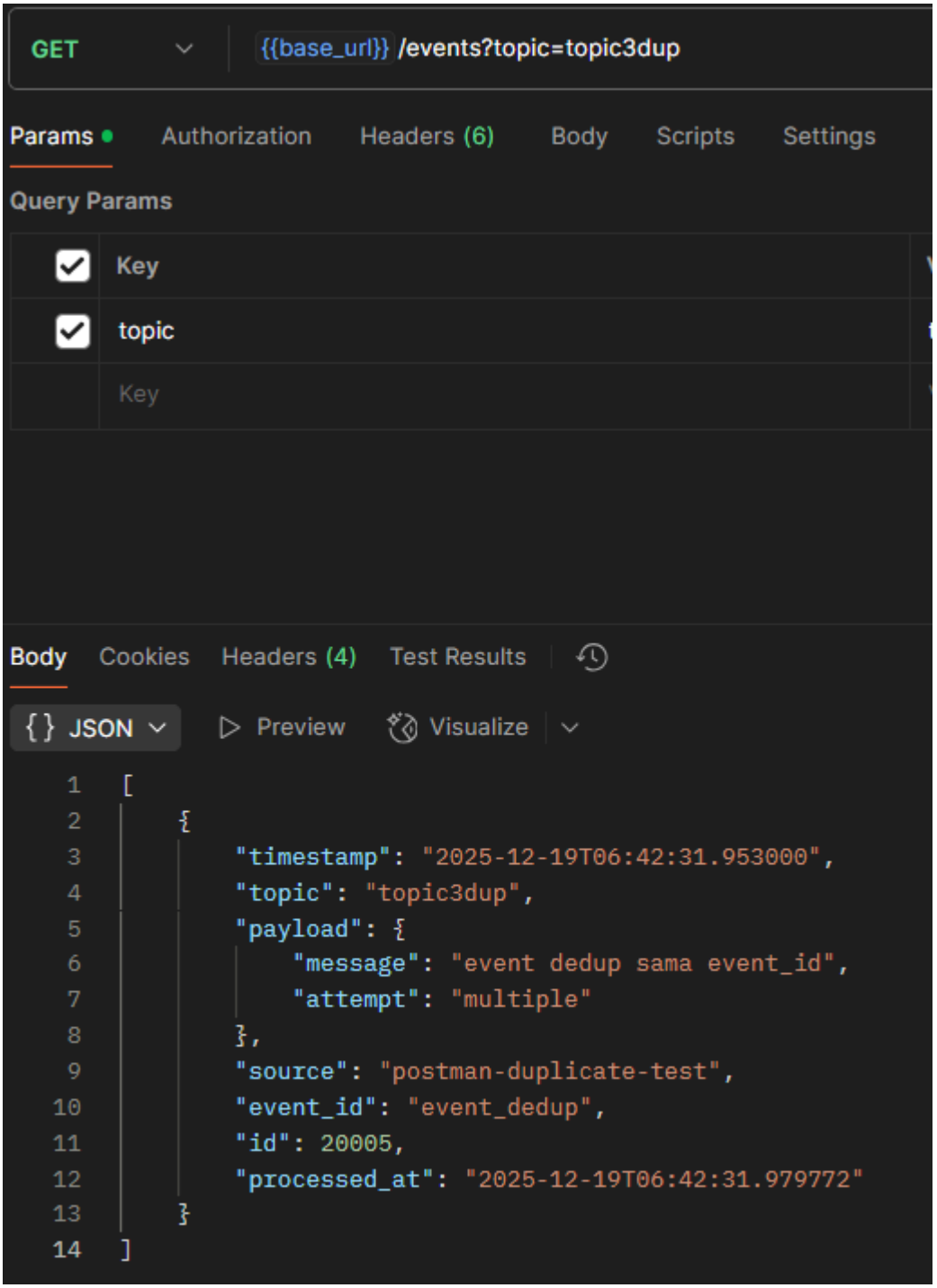
Disini kita coba memasukkan topic “topic3dup”



Pada /stats, received dan unique processed bertambah 1, duplicate masih sama.



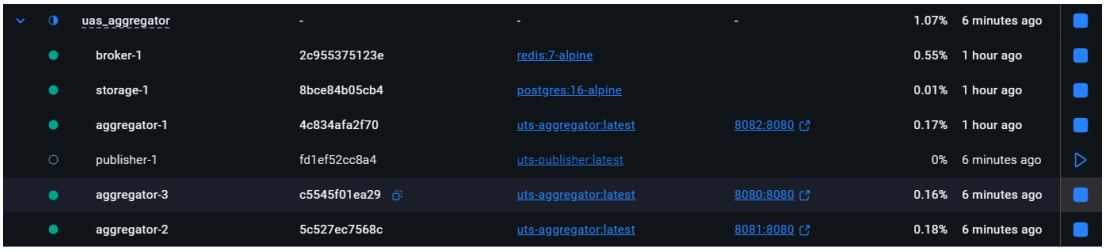
Kemudian saya publish lagi event yang sama dengan uuid yang sama 5 kali, hasilnya received dan duplicate bertambah 5, dan unique masih sama.



Dan event topic3dup yang tersimpan hanya 1.

d. Transaksi dan Konkurensi

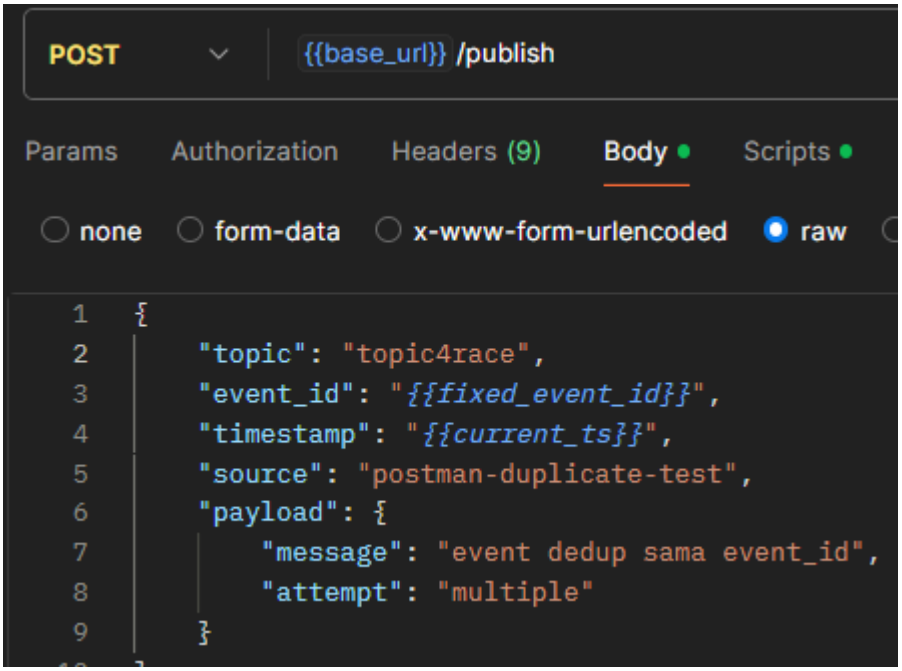
Disini kita akan mencoba 3 aggregator yang jalan bersamaan lalu jalankan transaksi beruntun menggunakan postman.



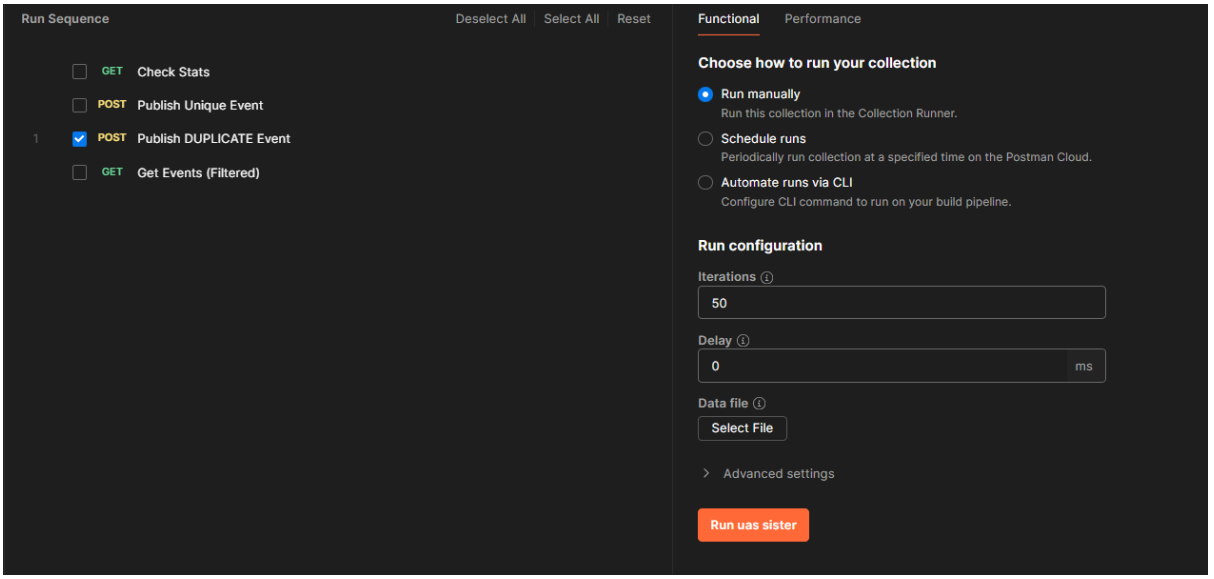
uas_aggregator	-	-	-	1.07%	6 minutes ago	
broker-1	2c955375123e	redis:7-alpine		0.55%	1 hour ago	
storage-1	8bce84b05cb4	postgres:16-alpine		0.01%	1 hour ago	
aggregator-1	4c834afa2f70	uts-aggregator:latest	8082:8080	0.17%	1 hour ago	
publisher-1	fd1ef52cc8a4	uts-publisher:latest		0%	6 minutes ago	
aggregator-3	c5545f01ea29	uts-aggregator:latest	8080:8080	0.16%	6 minutes ago	
aggregator-2	5c527ec7568c	uts-aggregator:latest	8081:8080	0.18%	6 minutes ago	


```
{
    "total_received_queued": 40010,
    "unique_processed_db": 36412,
    "estimated_duplicate_dropped": 3598,
    "topics_count": [
        {
            "topic": "topic2dup",
            "count": 1
        },
        {
            "topic": "login",
            "count": 9197
        },
        {
            "topic": "topic1",
            "count": 1
        },
        {
            "topic": "payment",
            "count": 9011
        },
        {
            "topic": "topic3dup",
            "count": 1
        },
        {
            "topic": "sensor",
            "count": 9037
        },
        {
            "topic": "order",
            "count": 9164
        }
    ],
```


/stats sebelum tes



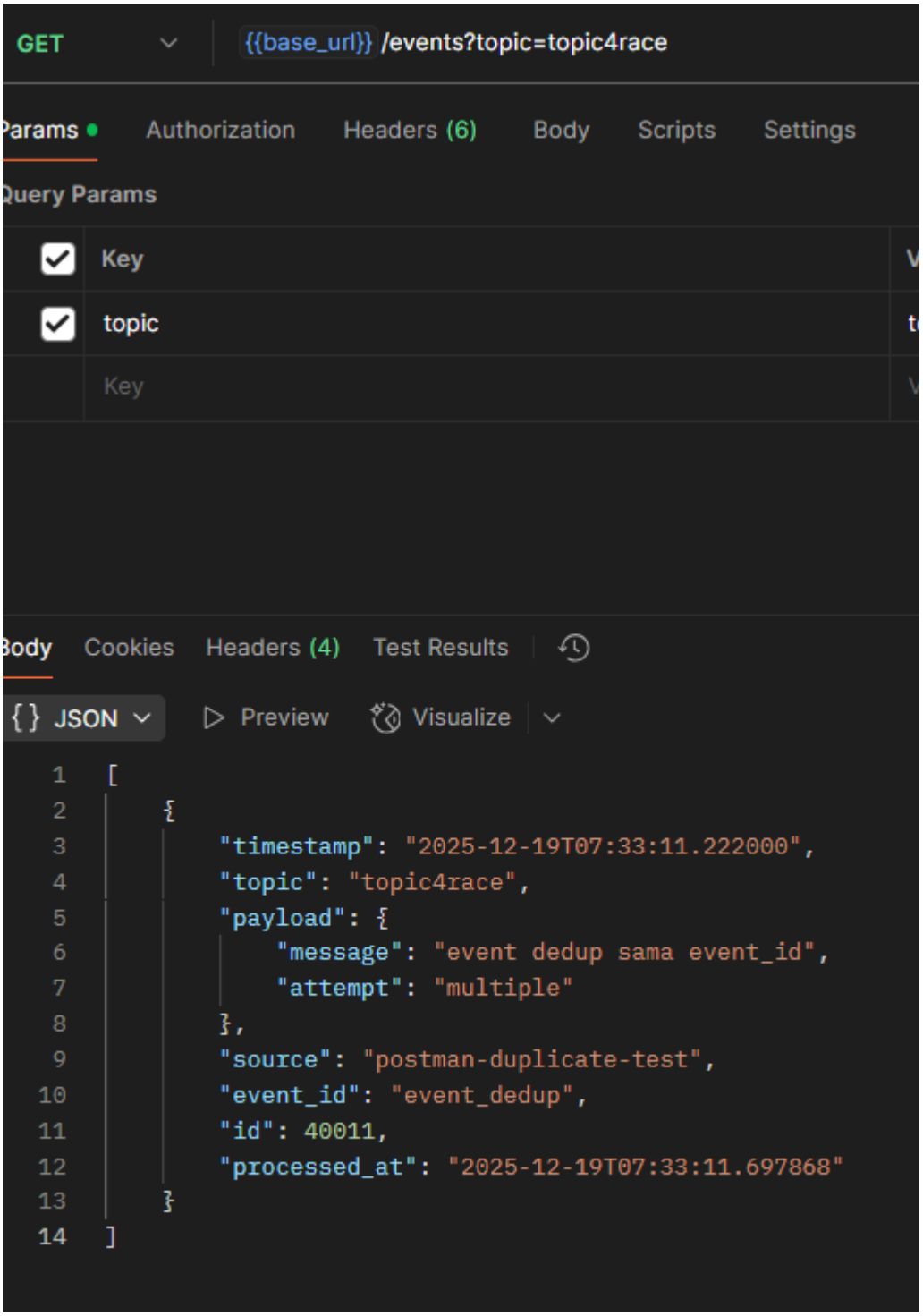
Nama topic: topic4race



Konfigurasi postman dengan 50 iterasi tanpa delay.

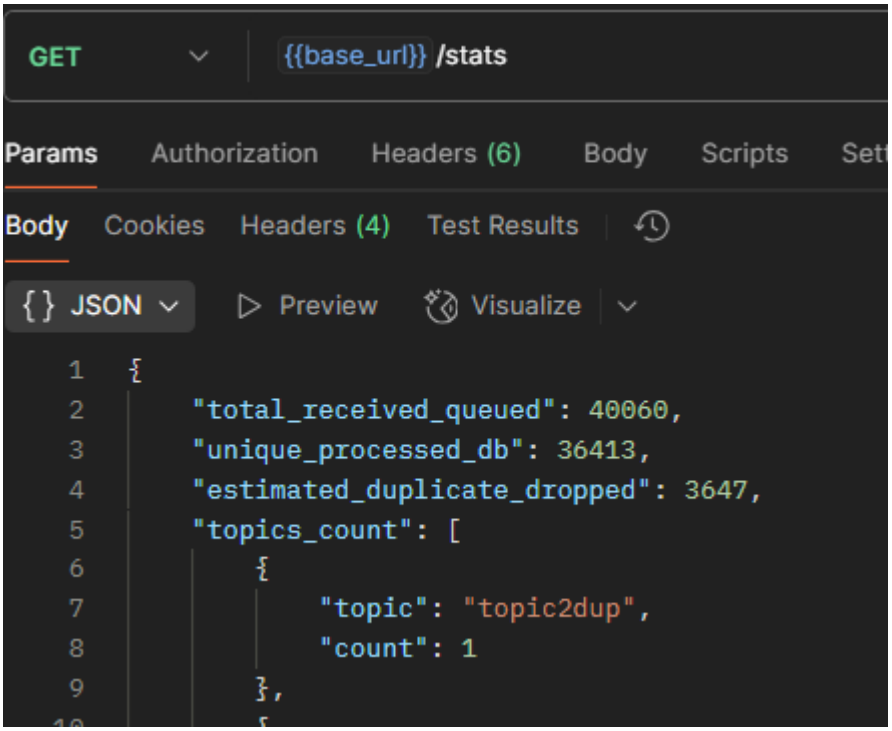
```
{
    "total_received_queued": 40060,
    "unique_processed_db": 36413,
    "estimated_duplicate_dropped": 3647,
    "topics_count": [
        {
            "topic": "topic2dup",
            "count": 1
        },
        {
            "topic": "login",
            "count": 9197
        },
        {
            "topic": "topic1",
            "count": 1
        },
        {
            "topic": "payment",
            "count": 9011
        },
        {
            "topic": "topic3dup",
            "count": 1
        },
        {
            "topic": "sensor",
            "count": 9037
        },
        {
            "topic": "topic4race",
            "count": 1
        },
        {
```


Hasil /stats setelah run 50 iterasi. Terlihat unique hanya naik 1, duplicate naik 47

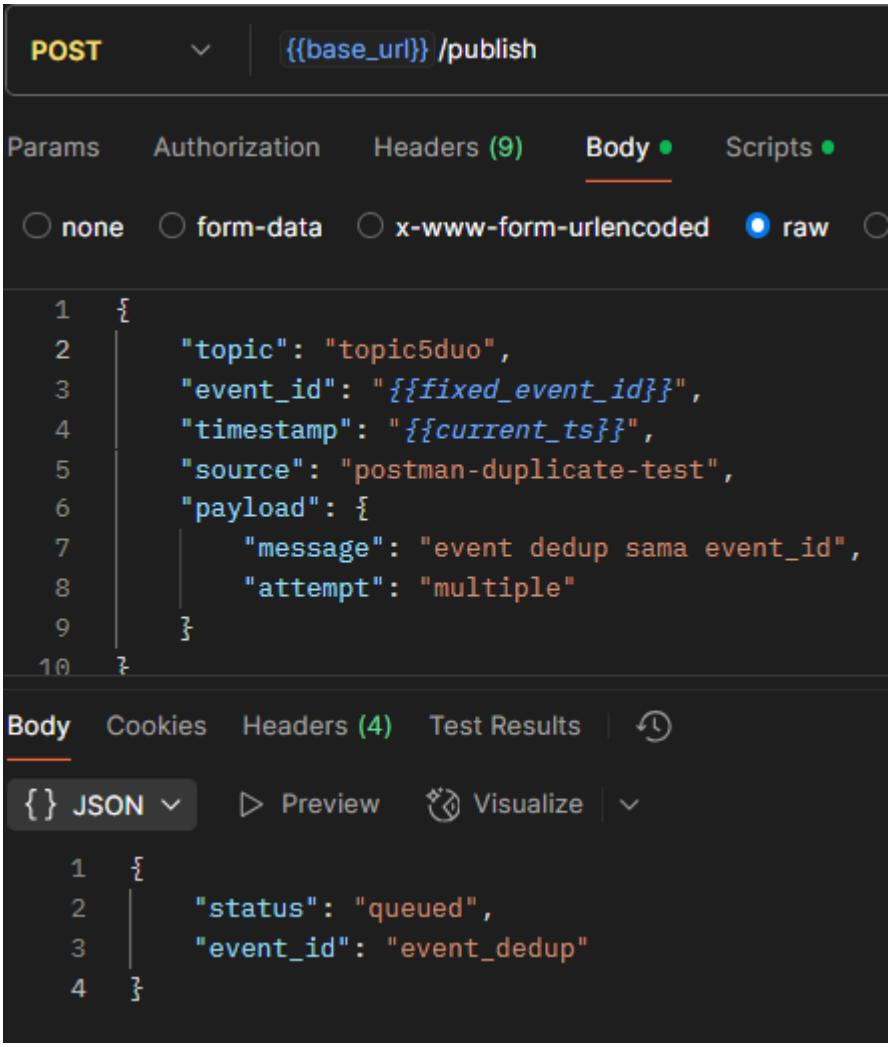


Hanya ada 1 event topic4race yang tersimpan.

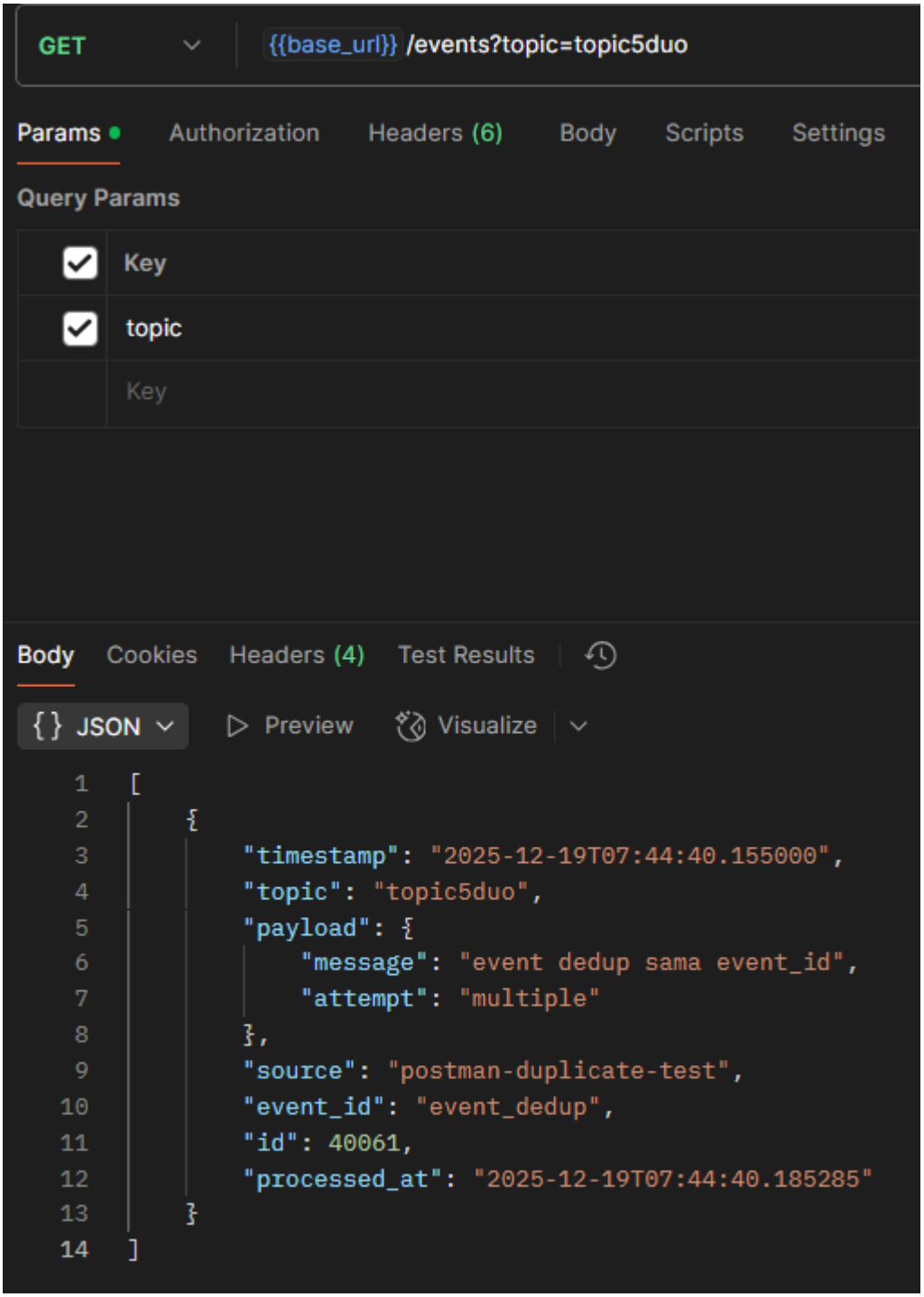
e. Reliability dan Ordering



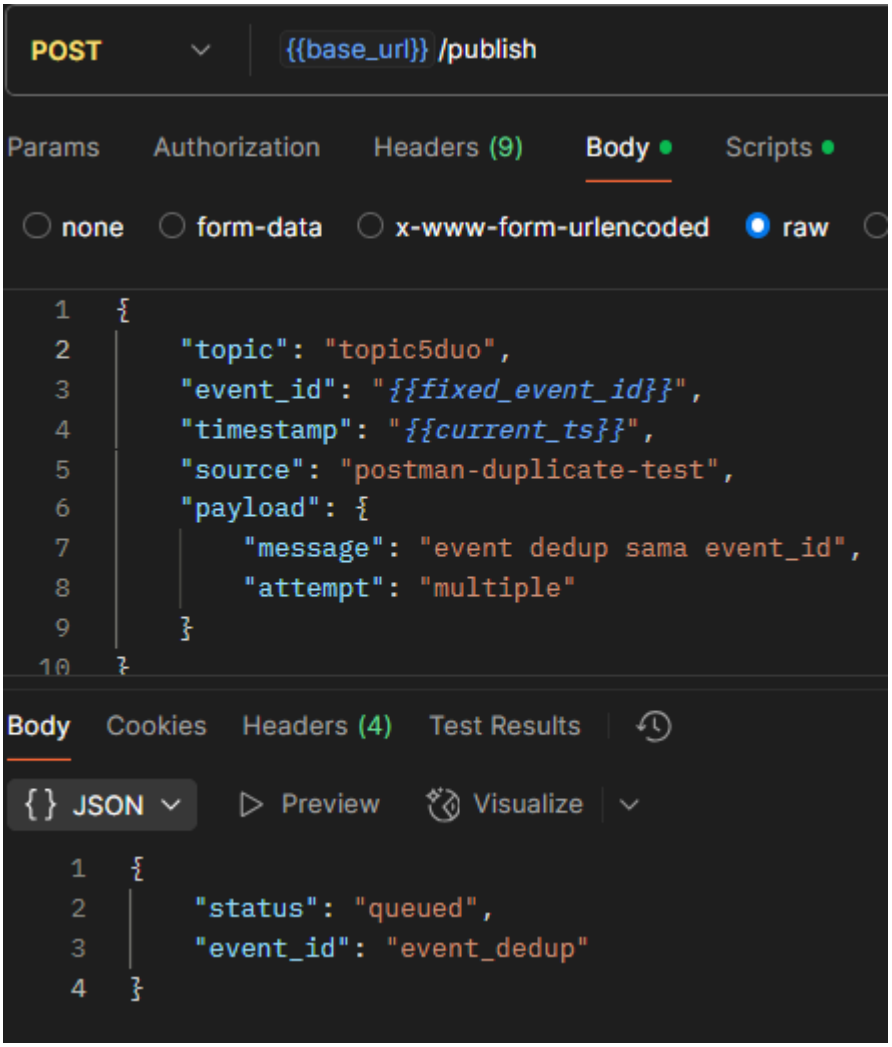
/stats sebelum testing.


POST
{{base_url}} /publish
Params
Authorization
Headers (9)
Body
Scripts
none
form-data
x-www-form-urlencoded
raw
{
"topic": "topic5duo",
"event_id": "{{fixed_event_id}}",
"timestamp": "{{current_ts}}",
"source": "postman-duplicate-test",
"payload": {
"message": "event dedup sama event_id",
"attempt": "multiple"
}
}
Body
Cookies
Headers (4)
Test Results
JSON
Preview
Visualize
{
"status": "queued",
"event_id": "event_dedup"
}

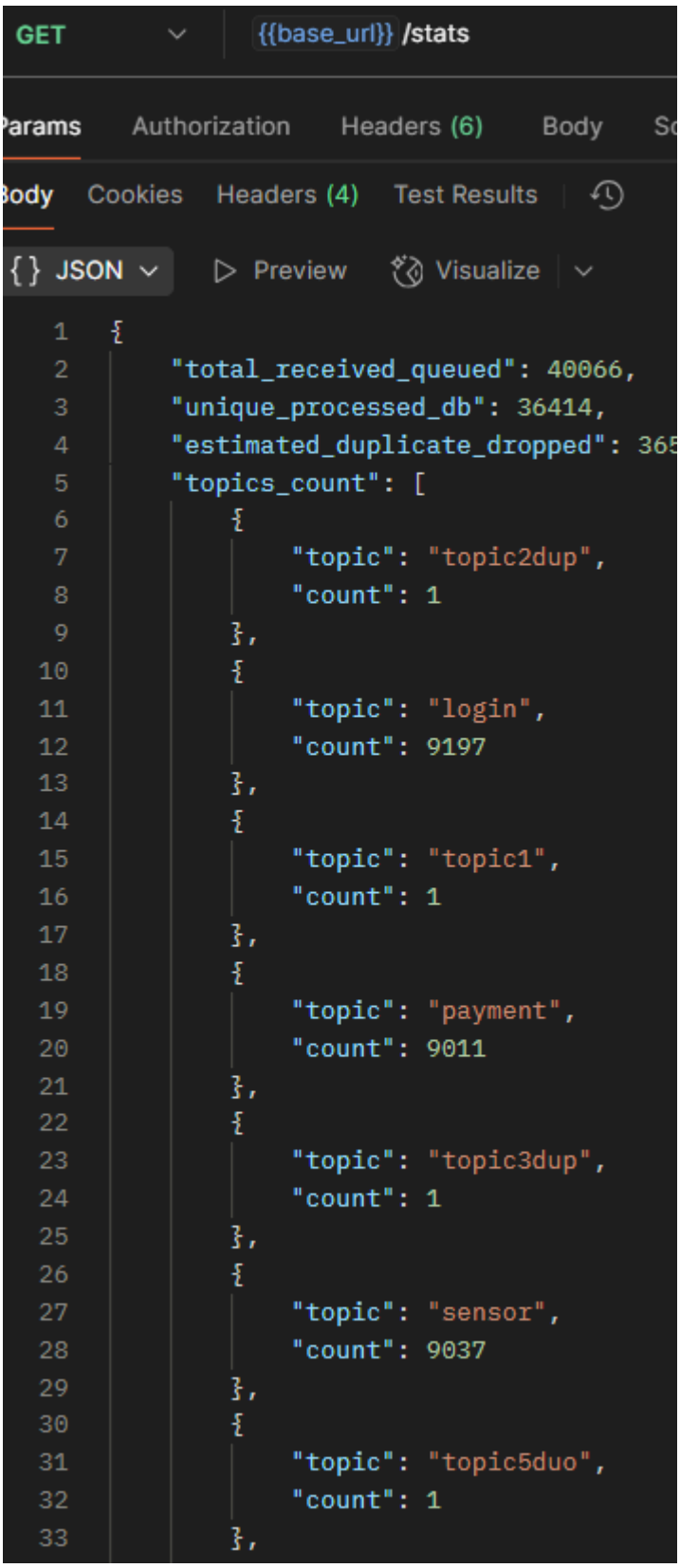
Setelah dikirim 1 kali, topic5duo tercatat dan stats received dan processed naik 1.



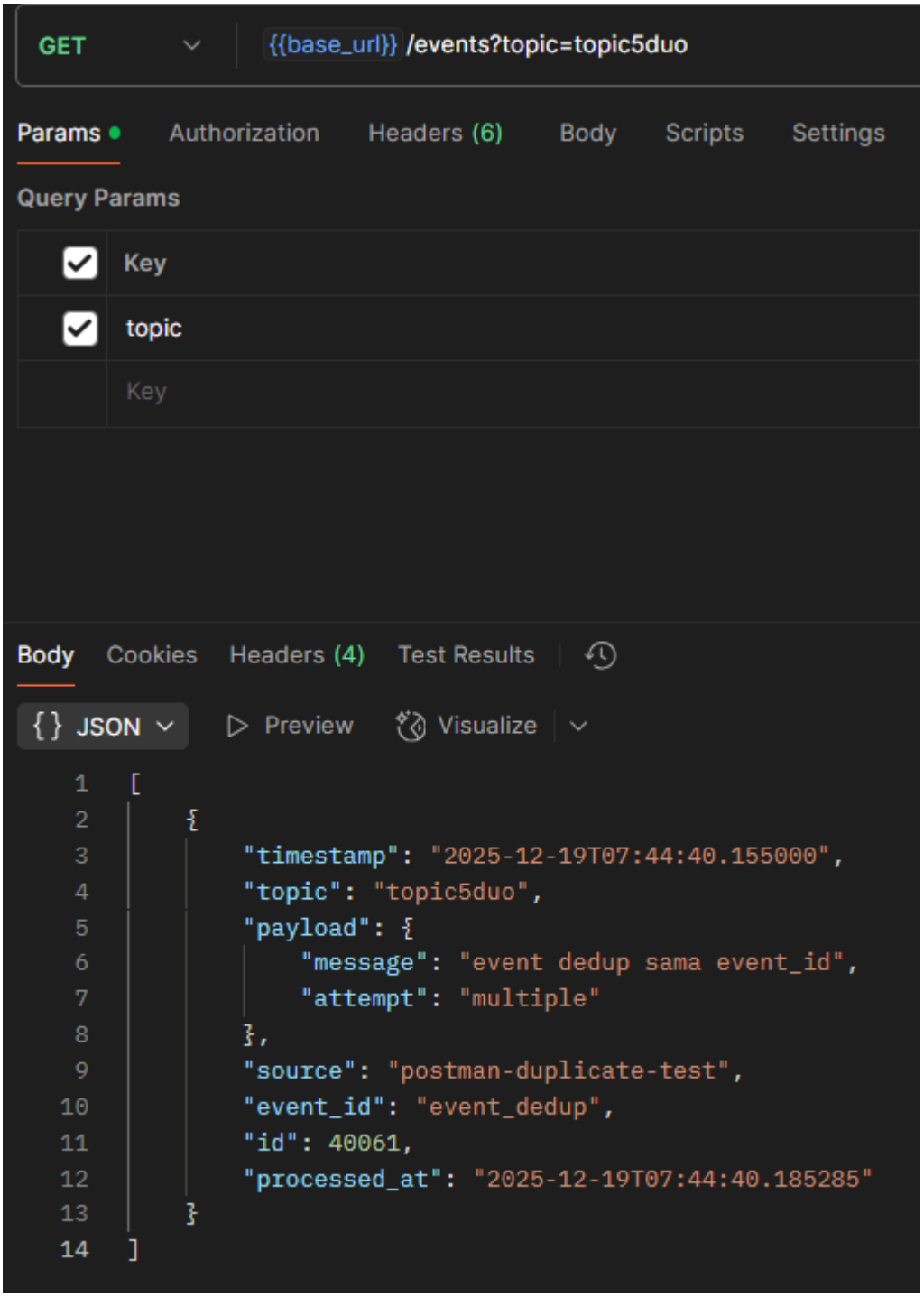
topic5duo hanya ada 1.



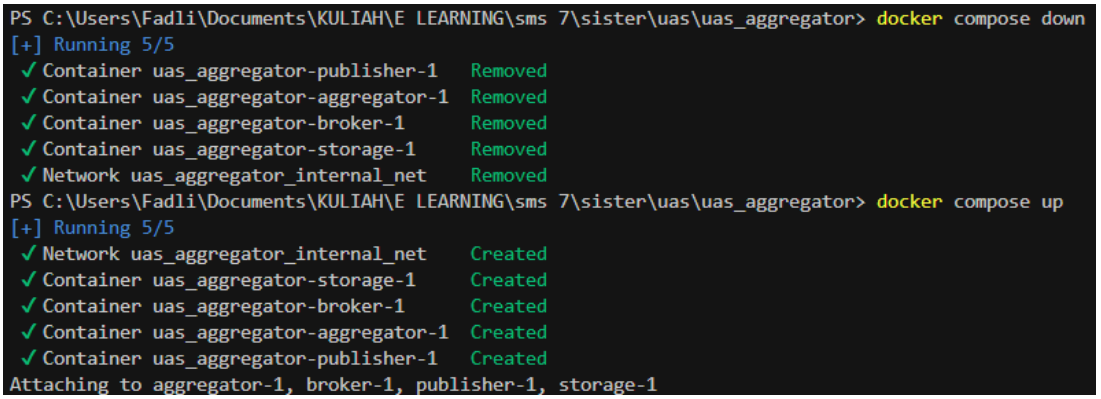
Sekarang event dan event_id yang sama dikirim hingga 5 kali.



Terlihat topic5duo yang tercatat masih 1 namun received dan duplicate naik 5 sedangkan processed masih sama.



Masih ada 1 topic5duo yang tersimpan.

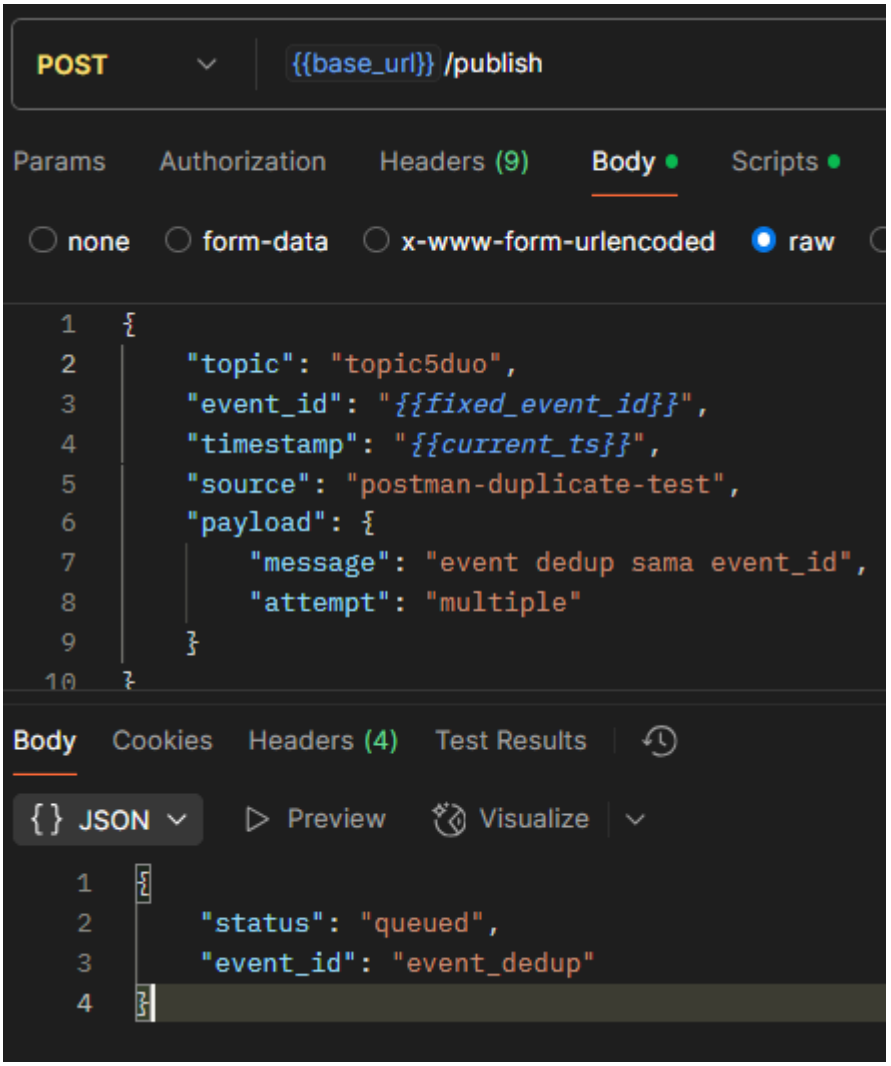


Untuk simulasi crash, container terlebih dahulu dihapus lalu dibuat ulang.


```
{
    "total_received_queued": 60066,
    "unique_processed_db": 54583,
    "estimated_duplicate_dropped": 5483,
    "topics_count": [
        {
            "topic": "login",
            "count": 13724
        },
        {
            "topic": "order",
            "count": 13773
        },
        {
            "topic": "payment",
            "count": 13581
        },
        {
            "topic": "sensor",
            "count": 13500
        },
        {
            "topic": "topic1",
            "count": 1
        },
        {
            "topic": "topic2dup",
            "count": 1
        },
        {
            "topic": "topic3dup",
            "count": 1
        },
        {
            "topic": "topic4race",
            "count": 1
        },
        {
            "topic": "topic5duo",
            "count": 1
        }
    ],
    "uptime_seconds": 1009.6286926269531
```

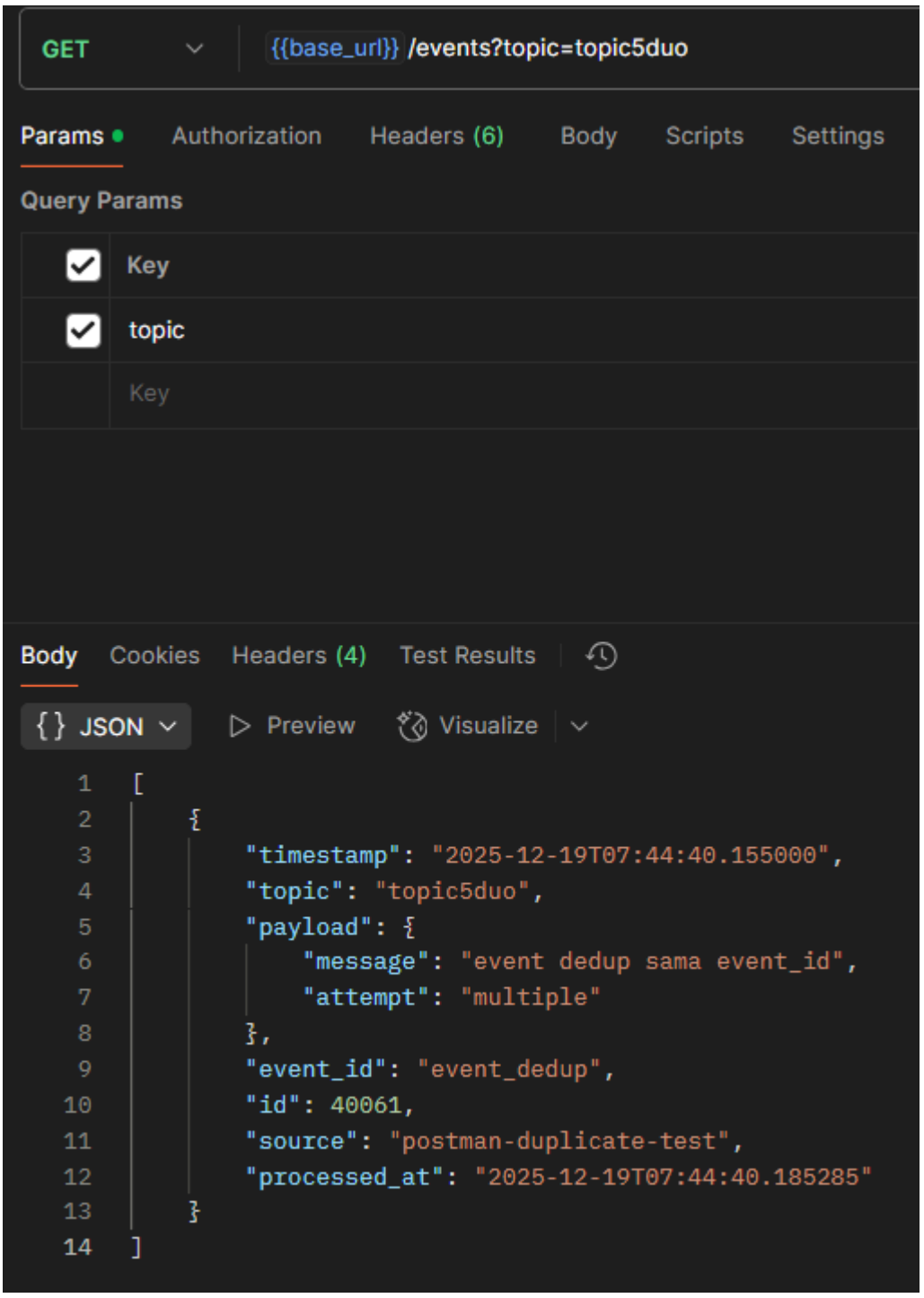
```
}
```

/stats setelah simulasi crash (container publisher jalan kembali menambahkan 20000 event) terlihat bahwa topic5duo masih ada.




```
{
    "total_received_queued": 60067,
    "unique_processed_db": 54583,
    "estimated_duplicate_dropped": 5484,
    "topics_count": [
        {
            "topic": "login",
            "count": 13724
        },
        {
            "topic": "order",
            "count": 13773
        },
        {
```

```
            "topic": "payment",
            "count": 13581
        },
        {
            "topic": "sensor",
            "count": 13500
        },
        {
            "topic": "topic1",
            "count": 1
        },
        {
            "topic": "topic2dup",
            "count": 1
        },
        {
            "topic": "topic3dup",
            "count": 1
        },
        {
            "topic": "topic4race",
            "count": 1
        },
        {
            "topic": "topic5duo",
            "count": 1
        }
    ],
    "uptime_seconds": 1834.587604045868
}
```



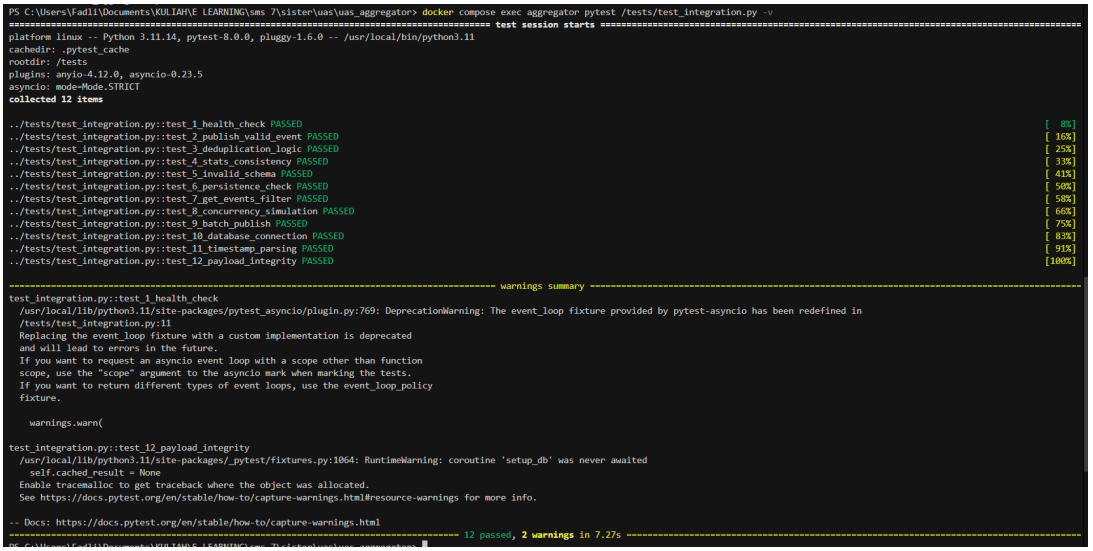
Event yang sama dikirimkan dan hasilnya topic5duo masih ada 1. Received dan duplicate naik 1 sedangkan processed masih sama.

Sistem tidak mengimplementasikan total ordering dikarenakan perannya sudah diambil alih oleh properti processed_at dan timestamp yang tercantum pada setiap data publish yang masuk dan properti ini dapat digunakan untuk mengurutkan data dengan konsisten tanpa mengorbankan performa.

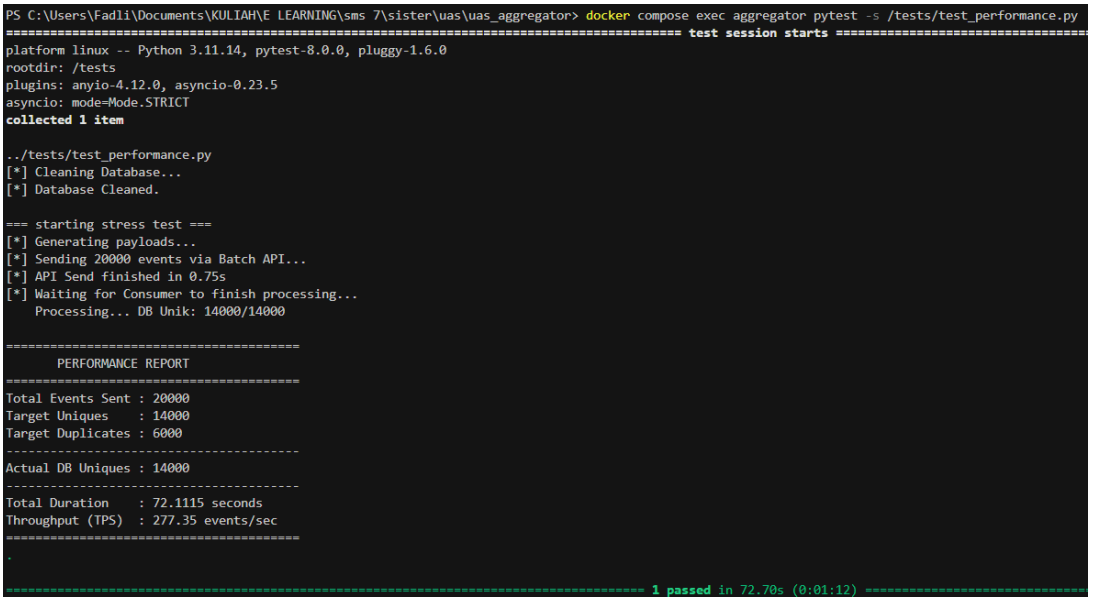
f. Performa Minimum

stress test dengan 20000 data dan 30% duplikat dengan hasil sebagai berikut:

1.



2. Single worker stress test



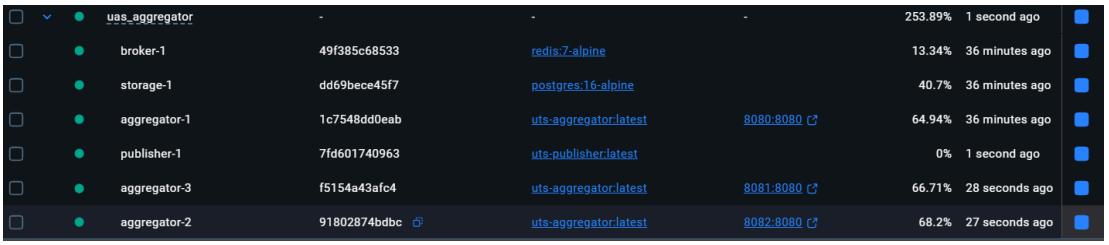
3. 3 worker integration test

```
PS C:\Users\Fadli\Documents\KULIAH\E LEARNING\sms 7\sister\uas\uas_aggregator> docker compose exec aggregator pytest /tests/test_integration.py -v
============================= test session starts ==============================
platform linux -- Python 3.11.14, pytest-8.0.0, pluggy-1.6.0 -- /usr/local/bin/python3.11
cachedir: .pytest_cache
rootdir: /tests
plugins: anyio-4.12.0, asyncio-0.23.5
asyncio: mode=Mode.STRICT
collected 12 items

../tests/test_integration.py::test_1_health_check PASSED                 [  8%]
../tests/test_integration.py::test_2_publish_valid_event PASSED          [ 16%]
../tests/test_integration.py::test_3_deduplication_logic PASSED          [ 25%]
../tests/test_integration.py::test_4_stats_consistency PASSED            [ 33%]
../tests/test_integration.py::test_5_invalid_schema PASSED               [ 41%]
../tests/test_integration.py::test_6_persistence_check PASSED            [ 50%]
../tests/test_integration.py::test_7_get_events_filter PASSED            [ 58%]
../tests/test_integration.py::test_8_concurrency_simulation PASSED       [ 66%]
../tests/test_integration.py::test_9_batch_publish PASSED                [ 75%]
../tests/test_integration.py::test_10_database_connection PASSED         [ 83%]
../tests/test_integration.py::test_11_timestamp_parsing PASSED           [ 91%]
../tests/test_integration.py::test_12_payload_integrity PASSED           [100%]

============================== warnings summary ===============================
test_integration.py::test_1_health_check
  /usr/local/lib/python3.11/site-packages/pytest_asyncio/plugin.py:769: DeprecationWarning: The event_loop fixture provided by pytest-asyncio has been redefined in
  /tests/test_integration.py:11
  Replacing the event_loop fixture with a custom implementation is deprecated
  and will lead to errors in the future.
  If you want to request an asyncio event loop with a scope other than function
  scope, use the "scope" argument to the asyncio mark when marking the tests.
  If you want to return different types of event loops, use the event_loop_policy
  fixture.

    warnings.warn(

test_integration.py::test_12_payload_integrity
  /usr/local/lib/python3.11/site-packages/_pytest/fixtures.py:1064: RuntimeWarning: coroutine 'setup_db' was never awaited
    self.cached_result = None
  Enable tracemalloc to get traceback where the object was allocated.
  See https://docs.pytest.org/en/stable/how-to/capture-warnings.html#resource-warnings for more info.

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
======================== 12 passed, 2 warnings in 7.50s ========================
```

4. 3 worker stress test

```
PS C:\Users\Fadli\Documents\KULIAH\E LEARNING\sms 7\sister\uas\uas_aggregator> docker compose exec aggregator pytest -s /tests/test_performance.py
============================= test session starts ==============================
platform linux -- Python 3.11.14, pytest-8.0.0, pluggy-1.6.0
rootdir: /tests
plugins: anyio-4.12.0, asyncio-0.23.5
asyncio: mode=Mode.STRICT
collected 1 item

../tests/test_performance.py
[*] Cleaning Database...
[*] Database Cleaned.

=== starting stress test ===
[*] Generating payloads...
[*] Sending 20000 events via Batch API...
[*] API Send finished in 0.97s
[*] Waiting for Consumer to finish processing...
    Processing... DB Unik: 14000/14000

========================================
       PERFORMANCE REPORT
========================================
Total Events Sent : 20000
Target Uniques    : 14000
Target Duplicates : 6000
----------------------------------------
Actual DB Uniques : 14000
----------------------------------------
Total Duration    : 27.1608 seconds
Throughput (TPS)  : 736.35 events/sec
========================================
.

============================== 1 passed in 27.72s ==============================
```

g. Docker dan Compose



Sistem memiliki 6 container terpisah yakni broker, storage, publisher, dan 3 aggregator.

```
services:
  aggregator:
    build: ./aggregator
    image: uts-aggregator:latest
    depends_on:
```

```
      storage:
        condition: service_healthy
      broker:
        condition: service_started
    environment:
      - DATABASE_URL=postgresql+asyncpg://user:pass@storage:5432/db
      - BROKER_URL=redis://broker:6379/0
    ports:
      - "8080-8082:8080"
    volumes:
      - ./tests:/tests
    networks:
      - internal_net

  publisher:
    build: ./publisher
    image: uts-publisher:latest
    depends_on:
      - aggregator
    environment:
      - TARGET_URL=http://aggregator:8080/publish
      - TOTAL_EVENTS=20000
    networks:
      - internal_net
    restart: on-failure

  broker:
    image: redis:7-alpine
    expose:
      - 6379
    volumes:
      - broker_data:/data
    networks:
      - internal_net

  storage:
    image: postgres:16-alpine
    environment:
      - POSTGRES_DB=db
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
    expose:
      - 5432
    volumes:
      - pg_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d db"]
      interval: 5s
      timeout: 5s
      retries: 5
    networks:
      - internal_net

volumes:
  pg_data:
  broker_data:

networks:
  internal_net:
    driver: bridge
```

Sistem dapat di build menggunakan docker compose dimana terdapat perintah

- Container aggregator yang bergantung dengan container storage dan broker dengan range port 8080-8082 (jika hanya 1 aggregator yang berjalan, akan diambil random diantaranya. Menggunakan network local.
- Container publisher bergantung dengan aggregator dan mengirimkan 20000 events dengan network local.
- Container broker menggunakan image redis pada port 6379 dengan volume broker_data pada network local.
- Container storage menggunakan image postgres pada port 5432 dengan volume pg_data pada network local.
- Network sepenuhnya local,

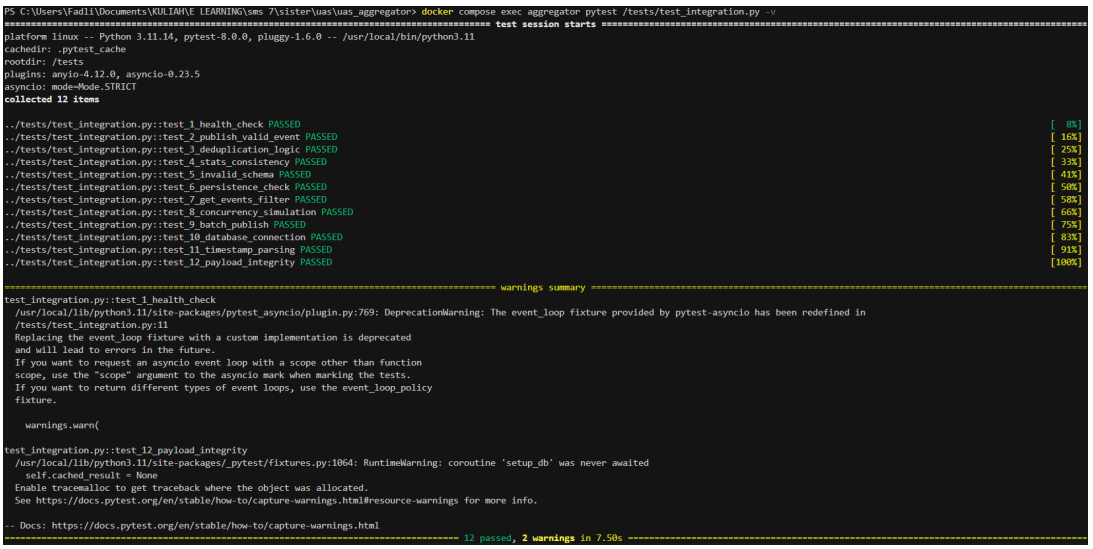
h. Unit test

Pada program terdapat 12 unit test yang menguji fungsionalitas sistem yakni:

1. **health_check** Tes ini berfungsi sebagai verifikasi berjalannya sistem dengan memanggil endpoint /stats. Tujuannya adalah memastikan bahwa service Aggregator telah berjalan di dalam container, API FastAPI menerima koneksi HTTP, dan tidak terjadi crash saat inisialisasi awal sebelum menjalankan tes logika selanjutnya.
2. **publish_valid_event** Tes ini menguji kemampuan API untuk menerima dan memproses satu event yang valid sesuai skema JSON yang ditentukan. Dengan mengirimkan payload lengkap ke endpoint /publish dan mengharapkan respons sukses ("queued"), tes ini memvalidasi bahwa komunikasi antara API Layer dan Message Broker (Redis) berfungsi dengan baik.
3. **deduplication_logic** Tes ini dijalankan dimana sebuah event dengan topic dan event_id yang sama dikirimkan sebanyak tiga kali berturut-turut. Tes ini memverifikasi bahwa meskipun request dikirim berulang kali, database PostgreSQL hanya menyimpan tepat satu baris data, membuktikan bahwa mekanisme ON CONFLICT DO NOTHING dan unique constraint berfungsi mencegah data ganda.
4. **stats_consistency** Tes ini memeriksa apakah endpoint /stats mengirim struktur data JSON yang valid dan nilai-nilai numerik yang konsisten. Tujuannya adalah memastikan bahwa sistem pemantauan internal mampu membaca counter dari Redis dan melakukan agregasi data dari Database tanpa error, serta memastikan field penting seperti unique_processed_db tidak bernilai null atau negatif.
5. **invalid_schema** Tes ini mensimulasikan pengiriman data yang tidak lengkap atau "cacat" (misalnya tanpa event_id) ke sistem untuk menguji ketahanan validasi input. Sistem harus menolak request tersebut dengan kode status HTTP 422 (Unprocessable Entity), memastikan bahwa hanya data yang mematuhi kontrak skema Pydantic yang boleh masuk ke dalam antrean pemrosesan.

6. **persistence_check** Tes ini melakukan query langsung ke database (SELECT) untuk memastikan bahwa data yang disimpan dari tes-tes sebelumnya masih tersedia dan tidak hilang. Dalam konteks pengujian integrasi yang melibatkan restart container, poin ini memvalidasi bahwa konfigurasi Volume Docker berfungsi dengan benar untuk menjaga keutuhan data.
7. **get_events_filter** Tes ini menguji kemampuan endpoint /events untuk mengambil data spesifik berdasarkan parameter query, seperti memfilter event berdasarkan topiknya. Tes ini memvalidasi bahwa query builder SQLAlchemy di backend mampu menerjemahkan parameter HTTP menjadi klausa WHERE SQL yang tepat, sehingga API hanya mengembalikan data yang relevan bagi pengguna.
8. **concurrency_simulation** Tes ini mengirimkan 50 request secara paralel dan hampir bersamaan menggunakan asyncio.gather untuk mensimulasikan kondisi high traffic atau race condition. Tujuannya adalah membuktikan ketangguhan sistem dalam menangani lonjakan beban mendadak tanpa mengalami deadlock, timeout, atau kegagalan koneksi pada API maupun Broker.
9. **batch_publish** Tes ini memverifikasi fungsionalitas endpoint /publish/batch dengan mengirimkan array berisi 10 event sekaligus dalam satu request HTTP. Tes ini memastikan bahwa implementasi Redis Pipeline di backend berjalan efektif, memungkinkan sistem memproses banyak data dengan latensi jaringan yang lebih rendah dibandingkan pengiriman satu per satu.
10. **database_connection** Tes ini menjalankan perintah SQL (SELECT 1) secara langsung melalui driver database, melewati lapisan logika aplikasi. Tujuannya adalah untuk mengecek koneksi jaringan internal Docker antara container Aggregator dan container Storage (PostgreSQL), memastikan tidak ada masalah autentikasi atau firewall internal.
11. **timestamp_parsing** Tes ini mengirimkan event dengan string waktu ISO-8601 tertentu, lalu mengambilnya kembali dari database untuk memeriksa properti tanggal dan jamnya. Ini berfungsi memvalidasi bahwa konversi tipe data dari String JSON menjadi objek DateTime Python (dan TIMESTAMP PostgreSQL) berjalan akurat, termasuk penanganan zona waktu (timezone) yang sering menjadi sumber bug.
12. **payload_integrity** Tes ini mengirimkan payload yang kompleks (berisi nested object atau array), kemudian memverifikasi bahwa data tersebut tersimpan utuh di kolom tipe JSONB database. Tujuannya adalah memastikan bahwa struktur data JSON tidak rusak, terpotong, atau terkonversi menjadi string biasa saat proses serialisasi dan deserialisasi antara API, Worker, dan Database.

Dua belas unit test berhasil dijalankan dalam 7.58 detik dengan hasil sebagai berikut:



Link video demo

https://youtu.be/GB_RH1ALuHg

Daftar Pustaka

van Steen, M., & Tanenbaum, A. S. (2023). *Distributed systems* (Edisi ke-4, Versi 01). distributed-systems.net. https://www.distributed-systems.net/index.php/books/ds4/ (distributed-systems.net)