

Pertemuan 5 — Arsitektur: *n-tier*, *Microservices*, dan *Messaging*

Dari satu aplikasi ke banyak layanan: alasan, pola, dan ongkosnya

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. *Tier dan layer: 1, 2, 3, sampai n -tier*
2. *SOA, dan mengapa ia ditinggalkan*
3. *Monolith, modular monolith, microservices*
4. *Conway's law dan kapan memecah layanan*
5. *API gateway dan service discovery*
6. *Synchronous vs asynchronous*
7. *RabbitMQ (queue) vs Kafka (log), pub-sub, event-driven*
8. *Transaksi lintas layanan: pola saga*
9. *Bertahan saat tetangga sakit: circuit breaker*
10. *Observability, lalu rancangan tugas besar UTS*

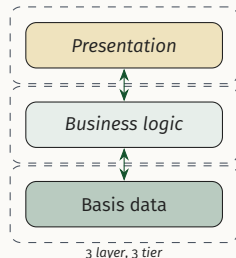
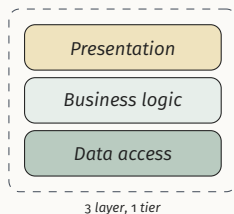
Sasaran pertemuan ini

Anda dapat membedakan 1-tier sampai n -tier, menimbang *monolith vs microservices*, menjelaskan *API gateway, service discovery*, dan *messaging*, serta menyusun rancangan arsitektur tugas besar UTS yang lengkap dengan alasan dan ongkosnya.

Tier dan layer

Layer itu logis, tier itu fisik

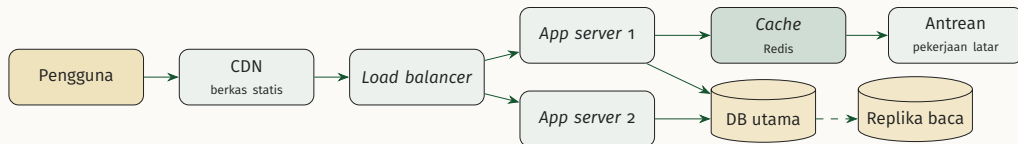
- **Layer** — pembagian **di dalam kode**: *presentation, business logic, data access*. Bisa saja semuanya berjalan dalam satu proses.
- **Tier** — pembagian **di mesin atau proses yang berbeda**, yang berkomunikasi lewat jaringan.
- Memindahkan satu *layer* ke *tier* sendiri berarti menambah satu panah jaringan: lebih fleksibel, lebih lambat, lebih banyak titik gagal.



Dari 1-tier sampai n -tier

Bentuk	Susunan	Contoh	Batasnya
1-tier	Tampilan, logika, data dalam satu program di satu mesin	Aplikasi desktop, Excel dengan makro	Tidak bisa dipakai bersama
2-tier	Klien tebal berisi tampilan dan logika; server basis data bersama	Aplikasi kasir ke DB pusat	Logika tersebar di tiap klien; pembaruan sulit
3-tier	Peramban, <i>application server</i> , basis data	Aplikasi web klasik	<i>App server</i> jadi titik skala utama
n -tier	Ditambah CDN, <i>load balancer</i> , <i>cache</i> , antrean, layanan pencarian	Hampir semua sistem web besar	Banyak komponen untuk dirawat

Wajah *n-tier* yang umum hari ini



- ▶ *App server* dibuat *stateless* supaya bisa ditambah sesuka hati di belakang *load balancer* (ingat REST di pertemuan 4).
- ▶ Basis data paling sulit diskalakan, karena itu ditopang *cache* dan replika baca.

Intinya

Ini masih **satu aplikasi** (*monolith*) yang dijalankan di banyak mesin. *N-tier* dan *microservices* adalah dua sumbu yang berbeda.

Service-Oriented Architecture (populer pada 2000-an) memecah sistem perusahaan menjadi layanan dengan kontrak resmi.

- ▶ Kontrak ditulis dalam WSDL, pesan dalam SOAP/XML
- ▶ Layanan dihubungkan lewat **Enterprise Service Bus** (ESB) yang juga melakukan *routing*, transformasi, dan orkestrasi
- ▶ Tujuannya pemakaian ulang layanan lintas unit bisnis

Mengapa terasa berat

- ▶ ESB menjadi pusat yang pintar dan rumit; perubahan apa pun harus lewat tim ESB
- ▶ XML dan standar WS-* yang panjang
- ▶ Layanan sering tetap berbagi satu basis data besar

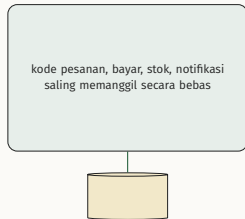
Intinya

Microservices sering dijelaskan sebagai SOA tanpa ESB raksasa: *smart endpoints and dumb pipes*. Logika ada di layanan; jalur komunikasinya dibuat sederhana.

***Monolith, modular monolith,
microservices***

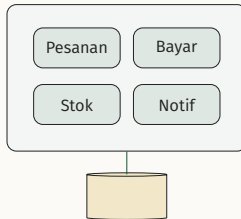
Tiga cara menyusun satu sistem

Monolith



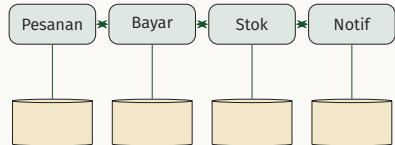
Satu *deploy*, satu basis data. Cepat dimulai; lama-lama batas antar bagian kabur (*big ball of mud*).

Modular monolith



Satu *deploy*, tetapi modul terbatas tegas dan hanya saling memanggil lewat API internal. Mudah dipecah kelak.

Microservices



Tiap layanan di-*deploy* sendiri dan **memiliki datanya sendiri**. Panggilan antar layanan lewat jaringan.

Ciri *microservices* yang sebenarnya

- ▶ **Satu layanan, satu kemampuan bisnis** — batasnya mengikuti *bounded context* (istilah dari *Domain-Driven Design*): pesanan, pembayaran, pengiriman.
- ▶ **Database per service** — layanan lain tidak boleh membaca tabel milik layanan ini secara langsung; semua lewat API atau *event*.
- ▶ **Independently deployable** — satu layanan bisa dirilis tanpa menunggu yang lain.
- ▶ **Dimiliki satu tim** — tim yang membangun juga yang menjalankan dan memperbaikinya (*you build it, you run it*).
- ▶ **Teknologi boleh berbeda** — Go untuk layanan yang butuh cepat, Python untuk ML, selama kontrak API-nya jelas.

Anti-pola: *distributed monolith*

Layanan sudah dipecah, tetapi masih berbagi basis data, harus dirilis bersamaan, dan satu permintaan memanggil sepuluh layanan berantai. Hasilnya: semua kerumitan *microservices*, tanpa satu pun manfaatnya.

Melvin Conway (1968) mengamati bahwa organisasi yang merancang sistem cenderung menghasilkan rancangan yang **meniru struktur komunikasi** organisasi itu sendiri.

- ▶ Tiga tim yang jarang bicara akan menghasilkan tiga komponen dengan antarmuka yang kaku di antaranya.
- ▶ Satu tim besar yang mengerjakan semua akan menghasilkan satu *monolith*.
- ▶ **Inverse Conway maneuver**: susun tim sesuai arsitektur yang Anda inginkan, maka arsitekturnya cenderung mengikuti.

Intinya

Microservices pada dasarnya jawaban atas masalah **organisasi**: bagaimana puluhan tim bisa bekerja dan merilis tanpa saling menunggu. Untuk satu tim berisi lima orang, masalah itu belum ada.

Kapan memecah, kapan jangan

Tanda sudah waktunya

- ▶ Banyak tim saling menunggu untuk merilis
- ▶ Satu bagian butuh skala jauh berbeda dari yang lain (misalnya pencarian vs admin)
- ▶ Satu bagian butuh keandalan atau keamanan lebih ketat (pembayaran)
- ▶ Batas domain sudah stabil dan dipahami

Tanda belum waktunya

- ▶ Tim kecil, produk masih mencari bentuk
- ▶ Belum ada otomatisasi *deploy*, *monitoring*, dan *tracing*
- ▶ Batas domain masih sering berubah
- ▶ Alasannya “karena perusahaan besar memakainya”

Intinya

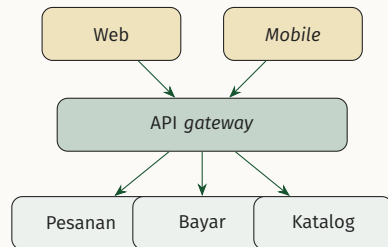
Jalur yang sering dianjurkan: mulai dari *modular monolith*, rapikan batas modulnya, lalu pecah modul yang paling jelas manfaatnya. Memecah terlalu dini lebih mahal daripada memecah terlambat.

Pintu masuk dan menemukan layanan

Satu pintu masuk untuk semua klien. Tugasnya:

- ▶ *Routing*: /pesanan/* ke layanan pesanan, dan seterusnya
- ▶ Autentikasi dan otorisasi di satu tempat (memeriksa token)
- ▶ *Rate limiting* dan perlindungan dari lonjakan
- ▶ *TLS termination*
- ▶ Agregasi: satu panggilan klien menjadi beberapa panggilan internal
- ▶ Penerjemahan: REST di luar, gRPC di dalam

Contoh: Kong, NGINX, Envoy, Traefik, layanan *gateway* milik penyedia *cloud*.



Backend for Frontend (BFF): satu *gateway* khusus per jenis klien, karena kebutuhan data web dan *mobile* sering berbeda.

Bahaya: *gateway* yang diberi logika bisnis akan menjadi ESB baru.

Service discovery: di mana layanan itu sekarang?

Di dunia *container*, alamat IP layanan berubah setiap kali ia dijalankan ulang atau diskalakan. Alamat tidak bisa ditulis mati di konfigurasi.

- ▶ **Service registry** — daftar layanan yang hidup beserta alamatnya (Consul, etcd, ZooKeeper). Layanan mendaftar saat hidup dan dihapus bila *health check*-nya gagal.
- ▶ **Client-side discovery** — klien bertanya ke *registry*, lalu memilih sendiri instans yang dipanggil.
- ▶ **Server-side discovery** — klien memanggil satu nama atau *load balancer*; yang memilih instans adalah infrastruktur.

Di Kubernetes

Layanan cukup dipanggil dengan nama DNS, misalnya `http://pembayaran:8080`. Objek Service Kubernetes meneruskan ke pod yang sehat. Ini *server-side discovery* yang sudah disediakan; kita bahas di pertemuan 7.

Konfigurasi dan secret

Konfigurasi (alamat, batas waktu) dan rahasia (kata sandi DB, kunci API) disimpan terpusat dan disuntikkan saat berjalan, **bukan** ditulis di kode atau repositori.

Komunikasi *asynchronous*

Synchronous vs asynchronous

Synchronous (REST, gRPC)

Pemanggil menunggu jawaban.

- ▶ Sederhana dan langsung tahu hasilnya
- ▶ **Temporal coupling**: penerima harus hidup **saat itu juga**
- ▶ Rantai $A \rightarrow B \rightarrow C$: *latency* dijumlahkan, *availability* dikalikan ($0,99^3 \approx 97\%$)

Asynchronous (message broker)

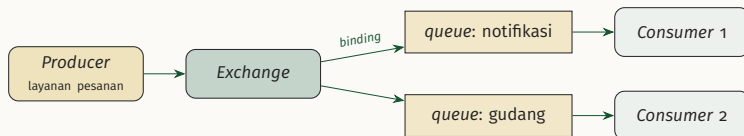
Pengirim menitipkan pesan, lalu melanjutkan pekerjaannya.

- ▶ Penerima boleh sedang mati; pesan menunggu di *broker*
- ▶ Meredam lonjakan: antrean menampung, penerima memproses sesuai kemampuannya
- ▶ Hasil tidak langsung diketahui; perlu cara melacak status

Intinya

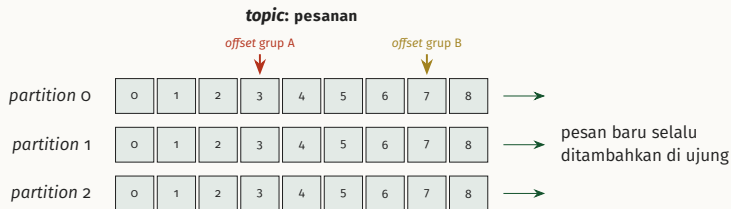
Aturan praktis: pakai *synchronous* bila pengguna **harus menunggu** jawabannya (cek saldo, harga). Pakai *asynchronous* untuk hal yang **boleh menyusul** (kirim notifikasi, perbarui laporan, kirim surel).

RabbitMQ: message queue



- ▶ *Producer* mengirim ke **exchange**; **binding** menentukan pesan masuk ke **queue** mana (*direct, topic, fanout*).
- ▶ *Consumer* mengambil pesan lalu mengirim **ack** setelah selesai. Bila *consumer* mati sebelum *ack*, pesan dikirim ulang ke *consumer* lain (*at-least-once*).
- ▶ Beberapa *consumer* pada satu *queue* berbagi beban (*competing consumers*). Pesan **dihapus** setelah di-*ack*.

Kafka: *distributed log*



- **Topic** dibagi menjadi **partition**; tiap *partition* adalah *log* yang hanya ditambah di ujung, dan direplikasi ke beberapa *broker*.
- Pesan **tidak dihapus** setelah dibaca; disimpan sesuai masa *retention*. Tiap **consumer group** mencatat **offset**-nya sendiri.
- Karena itu pesan bisa **dibaca ulang** (*replay*), dan banyak grup bisa membaca topik yang sama tanpa saling mengganggu.
- Urutan dijamin **di dalam satu partition**. Pesan dengan *key* yang sama (misalnya *id_pesanan*) masuk ke *partition* yang sama.

RabbitMQ atau Kafka?

	RabbitMQ (<i>queue</i>)	Kafka (<i>log</i>)
Model	Pesan dikirim ke penerima, lalu dihapus	Pesan disimpan; penerima membaca dengan <i>offset</i>
Baca ulang	Tidak (kecuali dirancang khusus)	Ya, selama masih dalam masa <i>retention</i>
Routing	Kaya: <i>direct</i> , <i>topic</i> , <i>fanout</i> , <i>header</i>	Sederhana: <i>topic</i> dan <i>key</i>
Kekuatan	Antrean tugas, <i>latency</i> rendah per pesan	<i>Throughput</i> sangat tinggi, aliran <i>event</i> , integrasi data
Contoh	Kirim surel, proses gambar, tugas latar	Jejak klik, <i>event</i> bisnis, CDC, <i>pipeline</i> data

Kafka dibahas jauh lebih dalam di mata kuliah Data Engineering (*streaming* dan *pipeline*). Di sini cukup memahami modelnya.

- ▶ **Publish-subscribe** — pengirim menerbitkan *event* ke topik tanpa tahu siapa penerimanya. Penerima berlangganan.
- ▶ **Event** — fakta yang **sudah terjadi**, ditulis dalam bentuk lampau: `PesananDibuat`, `PembayaranBerhasil`.
- ▶ Menambah penerima baru tidak perlu mengubah pengirim. Inilah *loose coupling*.

Dua gaya isi event

- ▶ **Event notification** — isinya hanya “pesanan 1001 dibuat”; penerima bertanya balik bila perlu detail
- ▶ **Event-carried state transfer** — *event* membawa data lengkap; penerima tidak perlu bertanya balik

Transactional outbox

Simpan perubahan data **dan** *event*-nya dalam satu transaksi DB (tabel *outbox*), lalu proses terpisah menerbitkannya ke *broker*. Mencegah data tersimpan tetapi *event* hilang.

Menjalankan RabbitMQ dengan Docker Compose

```
services:
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"    # AMQP
      - "15672:15672"  # dasbor web

  notifikasi:
    build: ./notifikasi
    environment:
      AMQP_URL: amqp://guest:guest@rabbitmq:5672/
    depends_on: [rabbitmq]
```

- ▶ Jalankan: `docker compose up -d`, lihat log dengan `docker compose logs -f rabbitmq`.
- ▶ Dasbor: `http://localhost:15672` (guest/guest, hanya boleh untuk lokal).
- ▶ Di dalam jaringan Compose, layanan lain memanggil *broker* dengan nama *service*-nya: `rabbitmq`. Ini bentuk paling sederhana dari *service discovery* berbasis DNS.
- ▶ Kata sandi sungguhan disimpan sebagai *secret*, bukan ditulis di berkas yang masuk repositori.

Menjalankan Kafka dengan Docker Compose

```
kafka:                                # satu broker, mode KRaft (tanpa ZooKeeper)
  image: apache/kafka:latest
  ports: ["9092:9092"]
  environment:
    KAFKA_NODE_ID: 1
    KAFKA_PROCESS_ROLES: broker,controller
    KAFKA_LISTENERS: PLAINTEXT://:9092,CONTROLLER://:9093
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
    KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
```

ADVERTISED_LISTENERS memakai nama kafka agar bisa dicapai dari *container* lain dalam jaringan Compose (dari laptop langsung, alamat ini perlu disesuaikan). Faktor replikasi 1 hanya untuk praktikum; di produksi minimal 3 *broker*. Sebaiknya kunci versi *image* (bukan `latest`) agar praktikum dapat diulang.

Mengirim dan menerima pesan (Python, pika)

Producer: layanan pesanan

```
import json, pika

conn = pika.BlockingConnection(
    pika.ConnectionParameters("localhost"))
ch = conn.channel()
ch.queue_declare(queue="pesanan.dibuat", durable=True)

ch.basic_publish(
    exchange="",
    routing_key="pesanan.dibuat",
    body=json.dumps({"id": 1001, "total": 25000}),
    properties=pika.BasicProperties(delivery_mode=2))
conn.close()
```

Consumer: layanan notifikasi

```
def tangani(ch, method, props, body):
    pesanan = json.loads(body)
    kirim_notifikasi(pesanan) # idempotent!
    ch.basic_ack(delivery_tag=method.delivery_tag)

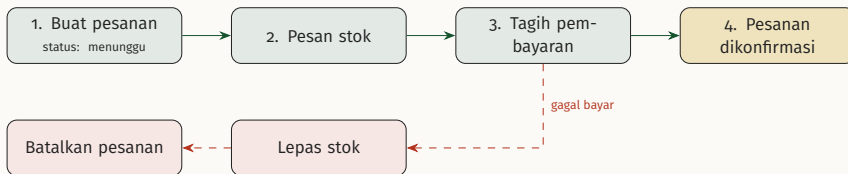
ch.queue_declare(queue="pesanan.dibuat", durable=True)
ch.basic_qos(prefetch_count=10)
ch.basic_consume(queue="pesanan.dibuat",
                  on_message_callback=tangani)
ch.start_consuming()
```

durable dan delivery_mode=2 membuat pesan bertahan bila broker dijalankan ulang. Ack dikirim **setelah** pekerjaan selesai.

Transaksi lintas layanan: *saga*

Tanpa satu basis data, tanpa satu transaksi

Pesanan menyentuh tiga layanan dengan tiga basis data. Tidak ada `BEGIN ... COMMIT` yang mencakup ketiganya. **Saga** memecahnya menjadi langkah lokal, masing-masing dengan **compensating action**.

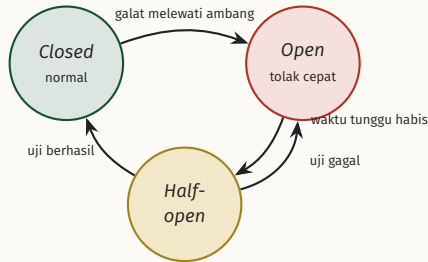


Choreography — tiap layanan bereaksi pada *event* layanan lain. Tanpa pusat; sederhana untuk sedikit langkah, sulit dilacak bila banyak.

Orchestration — satu *orchestrator* menyuruh tiap langkah dan menjalankan kompensasi. Alur terlihat jelas di satu tempat.

Bertahan saat tetangga sakit

Circuit breaker



- ▶ Saat layanan tujuan sering gagal, terus memanggilnya hanya membuang waktu dan memperparah bebannya.
- ▶ **Breaker open**: panggilan langsung ditolak tanpa menyentuh jaringan, dan pemanggil memakai cadangan (*fallback*): data *cache*, nilai bawaan, atau pesan “coba lagi nanti”.
- ▶ Setelah jeda, **half-open** mengizinkan beberapa panggilan uji.

Pola pendamping: **timeout** dan *retry* (pertemuan 4), **bulkhead** (memisahkan kolam koneksi per tujuan agar satu tujuan lambat tidak menghabiskan semuanya).

Intinya

Tujuannya mencegah *cascading failure*: satu layanan sakit tidak boleh menular ke seluruh sistem.

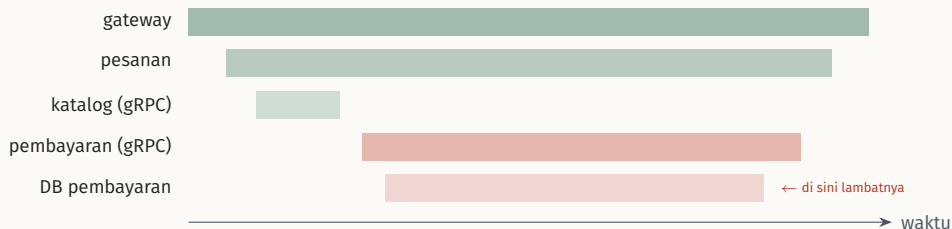
Observability

Tiga sinyal *observability*

Sinyal	Menjawab	Contoh	Alat umum
<i>Log</i>	Apa yang terjadi pada satu kejadian?	“pembayaran 1001 ditolak: saldo kurang”	Loki, Elasticsearch
<i>Metric</i>	Bagaimana tren secara agregat?	RPS, tingkat galat, P99 <i>latency</i>	Prometheus, Grafana
<i>Trace</i>	Ke mana saja satu permintaan pergi, dan di mana lambatnya?	Gateway 12 ms → pesanan 30 ms → bayar 180 ms	Jaeger, Tempo

- ▶ **OpenTelemetry** adalah standar terbuka untuk mengumpulkan ketiganya dari aplikasi, lalu mengirimkannya ke alat mana pun.
- ▶ *Log* sebaiknya terstruktur (JSON) dan memuat `trace_id` agar bisa dikaitkan dengan *trace*.
- ▶ Metrik ringkas yang berguna: **RED** untuk layanan (*Rate, Errors, Duration*) dan **USE** untuk sumber daya (*Utilization, Saturation, Errors*).

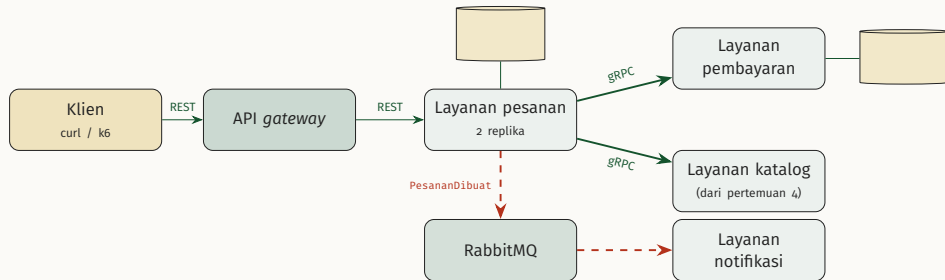
Distributed tracing



- ▶ Satu **trace** = satu permintaan dari ujung ke ujung. Tiap kotak adalah **span**: satu pekerjaan di satu layanan, dengan waktu mulai dan selesai.
- ▶ `trace_id` diteruskan di *header* setiap panggilan (standar W3C traceparent), sehingga semua *span* bisa dirangkai.
- ▶ Tanpa *tracing*, mencari penyebab lambat di sepuluh layanan adalah tebak-tebakan.

Merancang tugas besar UTS

Contoh rancangan: “Satu sistem, tiga cara bicara”



Garis tebal: *synchronous* (gRPC) karena pesanan butuh jawaban harga dan status bayar saat itu juga. Garis putus merah: *asynchronous*, karena notifikasi boleh menyusul dan tidak boleh menggagalkan pesanan.

Isi dokumen rancangan arsitektur UTS

1. Diagram arsitektur: layanan, basis data, *broker*, arah panah dan protokolnya
2. Daftar layanan: tanggung jawab, data yang dimiliki, teknologi
3. Kontrak API: *endpoint* REST (OpenAPI singkat) dan *.proto* gRPC
4. Jalur *asynchronous*: nama *event*, isi, siapa penerbit dan pelanggan
5. Skenario gagal: apa yang terjadi bila tiap layanan mati? *Timeout*, *retry*, *fallback* apa yang dipakai?
6. Rencana pengukuran: alat, jumlah pemanggil serentak, metrik yang dilaporkan
7. Alasan setiap keputusan, termasuk yang **tidak** dipilih

Intinya

Rancangan yang baik tidak hanya menunjukkan kotak dan panah, tetapi juga menjawab “mengapa panah ini *synchronous*?” dan “apa yang terjadi bila kotak ini mati?”.

Praktikum

Praktikum: menyalakan *broker* dan jalur *asynchronous* pertama

1. Buat `docker-compose.yml` berisi RabbitMQ (dan Kafka bila laptop kuat), lalu `docker compose up -d`.
2. Buka dasbor RabbitMQ di `localhost:15672`, amati *exchange* dan *queue*.
3. Jalankan *producer* dan *consumer* contoh. Kirim 1.000 pesan.
4. Eksperimen ketahanan:
 - Matikan *consumer*, kirim 100 pesan, lalu nyalakan lagi. Apakah semua pesan diproses?
 - Hentikan *consumer* **sebelum** *ack*. Apa yang terjadi pada pesan itu?
 - Jalankan dua *consumer* sekaligus. Bagaimana pesan dibagi?
5. (Kafka) Buat topik dengan 3 *partition*, kirim pesan ber-*key*, baca dengan dua *consumer group* berbeda, lalu ulangi pembacaan dari *offset* awal.
6. Mulai susun dokumen rancangan arsitektur UTS dalam kelompok.

Tugas untuk pertemuan berikutnya

1. Kumpulkan **dokumen rancangan arsitektur UTS** (tujuh bagian pada slide sebelumnya), maksimal empat halaman ditambah diagram.
2. Laporan praktikum: hasil tiga eksperimen ketahanan RabbitMQ beserta penjelasannya.
3. Jawab singkat: pada sistem Anda, panah mana yang akan Anda ubah menjadi *asynchronous* bila beban naik sepuluh kali lipat? Mengapa?
4. Pastikan Go, JDK 17 atau lebih baru, dan sbt (untuk Scala/Akka) terpasang di laptop.

Pertemuan depan

Di dalam setiap layanan pun ada banyak hal yang berjalan bersamaan: *thread* dan *lock*, *goroutine* dan *channel*, *actor model*, *reactive streams*, dan *async*. Kita bandingkan semuanya pada satu soal yang sama.

Terima kasih

Pertanyaan?