Nama : Dhea Safira

Kelas : TK 2A

Prodi : D3 Teknik Komputer

Mata Kuliah : Pemrograman Microservice

Ringkasan mengenai materi VIDIO 14 Backend for Frontend:

Video ini menjelaskan konsep "Backend for Frontend" (BFF) dalam pengembangan aplikasi berbasis microservices. Berikut ringkasannya:

Permasalahan dalam Microservices dengan Berbagai Frontend

1. Beragam Frontend: Aplikasi modern sering kali memiliki beberapa jenis frontend seperti website, mobile app, dan integrasi dengan third-party.
2. Autentikasi Berbeda: Setiap jenis frontend memerlukan metode autentikasi yang berbeda.
3. Kebutuhan Bandwidth Berbeda: Setiap frontend memiliki kebutuhan bandwidth yang berbeda, sehingga tidak bisa diberikan data yang sama besarnya.
4. API Berbeda: Data yang dibutuhkan tiap frontend berbeda, sehingga API yang digunakan juga berbeda.

Backend for Frontend (BFF)

- Konsep BFF: Membuat backend khusus untuk setiap jenis frontend (misalnya, satu untuk website, satu untuk mobile, dan satu untuk third-party).
- Keuntungan BFF:
- Pengembangan Terisolasi: Pengembangan backend dapat terisolasi untuk setiap frontend.
- Logika Tidak Tercampur: Logika untuk setiap frontend tidak tercampur dalam satu tempat.
- Pemeliharaan Lebih Mudah: Tim pengembangan dapat fokus pada satu frontend tanpa mengganggu yang lain.

Alternatif: GraphQL

- Konsep GraphQL: Sebuah query language untuk API yang memungkinkan klien menentukan data apa saja yang mereka butuhkan.

Keuntungan GraphQL:

- Kebebasan Klien: Frontend dapat menentukan data apa saja yang diperlukan.

Respon yang Dapat Disesuaikan: Backend dapat menyediakan data lengkap, dan klien dapat memilih bagian data yang dibutuhkan.

- Kekurangan GraphQL:

Kompleksitas Pengembangan: Butuh pengembangan tambahan untuk server dan klien GraphQL.

- Pusat Kesalahan: Jika ada masalah pada server GraphQL, semua klien akan terpengaruh.

Pilihan Tergantung Kebutuhan: Jika tim pengembang besar dan aplikasi kompleks, BFF mungkin lebih baik. Jika tim kecil dan ingin fleksibilitas, GraphQL bisa menjadi pilihan yang lebih baik.

Pertimbangan Implementasi: Implementasi tergantung pada kompleksitas dan kebutuhan spesifik dari proyek.

Selanjutnya

Penjelasan akan berlanjut ke topik "CQRS" (Command Query Responsibility Segregation) pada sesi berikutnya.

Ringkasan mengenai materi VIDIO 15 CQRS (Command Query Responsibility Segregation):

Video ini membahas konsep **CQRS (Command Query Responsibility Segregation)** yang memisahkan operasi *command* dan *query* dalam aplikasi. Berikut adalah poin-poin utama yang dijelaskan dalam video:

1. **Definisi CQRS**: CQRS adalah pola yang memisahkan operasi *command* (yang mengubah data seperti create, update, delete) dan *query* (yang mengambil data seperti read).
2. **Penerapan di Microservices**: Dalam arsitektur microservices, tiap layanan sebaiknya memiliki database sendiri. Saat data aplikasi semakin besar dan query mulai lambat, CQRS membantu dengan memisahkan database untuk operasi *command* dan *query*.
3. **Keuntungan CQRS**:
    - **Optimasi Kinerja**: Dengan memisahkan operasi, masing-masing database bisa dioptimasi sesuai kebutuhannya. Misalnya, *command* menggunakan database relasional seperti MySQL, sedangkan *query* menggunakan database yang dioptimasi untuk pencarian seperti Elasticsearch.

    - **Pengembangan Lebih Mudah**: Tim yang berbeda dapat mengerjakan modul *command* dan *query* secara terpisah, sesuai keahlian masing-masing.
    - **Skalabilitas**: Aplikasi bisa lebih mudah diskalakan karena beban kerja antara operasi *command* dan *query* bisa dipisahkan.
4. **Implementasi CQRS**:
    - **Messaging**: Menggunakan broker pesan untuk komunikasi antar layanan, meningkatkan ketahanan aplikasi jika salah satu layanan mengalami masalah.
    - **Real-time vs Non Real-time**: Menggunakan database yang berbeda untuk kebutuhan real-time (update cepat) dan non real-time (query cepat) memberikan fleksibilitas dan performa optimal.
5. **Praktik Terbaik**:
    - **Backup dan Recovery**: Data dari *command* bisa disimpan di messaging broker, memungkinkan recovery yang mudah jika layanan *query* mengalami downtime.
    - **Modularitas Kode**: Memisahkan modul *command* dan *query* dalam kode meningkatkan keterbacaan dan pemeliharaan kode.

Konsep CQRS sangat berguna dalam mengoptimasi aplikasi, terutama yang berbasis microservices dengan volume data besar. Video ini juga memberikan contoh bagaimana mengimplementasikan CQRS dalam aplikasi nyata dengan menggunakan messaging broker untuk meningkatkan ketahanan dan performa aplikasi.

Ringkasan mengenai materi VIDIO 16 Server Side Discovery:

Video tersebut membahas tentang konsep **server-side discovery** dalam konteks arsitektur microservices, menguraikan tantangan dan solusi yang terkait dengan manajemen node serta service discovery. Berikut adalah kesimpulan detail dari isi video:

- **High Availability**: Dalam production, setiap service biasanya memiliki lebih dari satu node untuk memastikan ketersediaan tinggi (high availability). Jika satu node mati, node lain dapat mengambil alih.
- **Masalah Lokasi Dinamis**: Jumlah node untuk setiap service dapat berubah secara dinamis berdasarkan beban trafik, yang menyebabkan perubahan alamat IP atau hostname. Hal ini menimbulkan tantangan dalam mengakses service yang benar.

- **Service Discovery**: Bagaimana cara mengetahui lokasi node service yang tepat ketika node-node tersebut bisa bertambah atau berkurang?

Solusi: Server-Side Discovery

- **Konsep Dasar**: Menggunakan load balancer atau proxy server di depan service untuk mengarahkan permintaan (requests) ke node-node yang tepat. Klien hanya perlu tahu lokasi load balancer, bukan lokasi node-node service.
- **Keuntungan**:
    - Klien tidak perlu tahu detail tentang node-node service yang dinamis.
    - Load balancer akan membagi beban trafik secara seimbang ke node-node service.
- **Proses**:

    - Misalnya, sebuah service bernama `product-service` memiliki tiga node. Load balancer akan mendistribusikan permintaan ke salah satu dari tiga node tersebut.
    - Jika node bertambah atau berkurang, hanya konfigurasi load balancer yang perlu diubah, bukan konfigurasi klien.

- **Alat Load Balancer Populer**:
    - NGINX
    - HAProxy
    - Traefik
    - Consul
- **Contoh Penggunaan**:
    - Sebuah `logistics-service` memiliki dua node dan satu load balancer. Jika service lain ingin berkomunikasi dengan `logistics-service`, mereka mengirim permintaan ke load balancer, bukan langsung ke node.

- **Jumlah Load Balancer**: Setiap service membutuhkan load balancer sendiri untuk menghindari single point of failure. Jika satu load balancer melayani banyak service, maka akan menjadi bottleneck dan berpotensi menjadi titik kegagalan tunggal.
- **Biaya dan Kompleksitas**: Implementasi ini memerlukan banyak instance load balancer, yang meningkatkan biaya dan kompleksitas pengelolaan infrastruktur.

- **Sederhana Tapi Mahal**: Server-side discovery adalah solusi yang sederhana dan efektif untuk memastikan service discovery dalam arsitektur microservices. Namun, solusi ini mahal karena membutuhkan banyak load balancer untuk setiap service.
- **Kebutuhan High Availability**: Untuk menghindari single point of failure, setiap service biasanya memiliki lebih dari satu instance load balancer, yang semakin menambah biaya.
- **Pertimbangan**: Meskipun mahal, pendekatan ini sangat andal dan umum digunakan dalam perusahaan besar untuk memastikan ketersediaan dan skalabilitas layanan mereka.

- Video ini juga menyebutkan bahwa ada alternatif lain yang akan dibahas selanjutnya, yaitu **client-side discovery**, yang akan memberikan pendekatan berbeda untuk masalah service discovery dalam arsitektur microservices.

Dengan demikian, video tersebut memberikan wawasan mendalam tentang bagaimana server-side discovery bekerja, keuntungannya, kekurangannya, dan beberapa pertimbangan praktis dalam mengimplementasikannya di lingkungan produksi.

Ringkasan mengenai materi VIDIO 17 Client Side Discovery :

Video tersebut membahas tentang konsep **client-side discovery** dalam arsitektur microservices sebagai alternatif dari server-side discovery. Berikut adalah kesimpulan dari penjelasan dalam video:

- **Kekurangan Server-Side Discovery**: Membutuhkan banyak load balancer untuk tiap service yang meningkatkan biaya dan kompleksitas. Untuk menghindari single point of failure, perlu lebih dari satu load balancer per service.

- **Client-Side Discovery**: Solusi yang menghilangkan kebutuhan akan load balancer, dengan memindahkan logika distribusi trafik ke klien.

Konsep Client-Side Discovery

- **Prinsip Dasar**: Klien harus mengetahui lokasi semua node service yang ingin diakses. Setiap klien mengirimkan permintaan (request) langsung ke node service, tanpa melalui load balancer.
- **Distribusi Trafik**: Klien bertanggung jawab mendistribusikan trafik ke node-node service menggunakan algoritma seperti round-robin atau lainnya.

- **Pengaturan Endpoint**: Klien harus diatur untuk mengetahui semua endpoint dari node-node service yang dituju.
- **Contoh**: Jika service `merchant-service` ingin mengakses `product-service` yang memiliki empat node, `merchant-service` harus tahu lokasi keempat node tersebut dan mendistribusikan trafik di antara mereka.
- **Libraries**: Beberapa bahasa pemrograman menyediakan library untuk memudahkan implementasi ini, seperti Netflix Ribbon untuk Java.

- **Biaya Lebih Rendah**: Tidak membutuhkan banyak hardware untuk load balancer, mengurangi biaya infrastruktur.
- **Kontrol Distribusi**: Memberikan fleksibilitas penuh kepada klien untuk mengimplementasikan logika distribusi trafik sesuai kebutuhan.

- **Kompleksitas di Klien**: Klien harus tahu lokasi semua node service, yang menambah kompleksitas konfigurasi dan pengelolaan.
- **Pembaruan Dinamis**: Jika ada perubahan jumlah node (bertambah atau berkurang), klien harus diperbarui dengan informasi lokasi baru, yang bisa jadi rumit dan memerlukan restart atau pembaruan konfigurasi.
- **Distribusi Tidak Merata**: Jika klien salah mengimplementasikan logika distribusi trafik, bisa menyebabkan ketidakseimbangan beban pada node-node service.

- **Pemilihan Pendekatan**: Kedua metode, server-side dan client-side discovery, memiliki kelebihan dan kekurangan masing-masing. Pilihan tergantung pada anggaran dan kebutuhan infrastruktur.
- **Saran**: Jika biaya bukan masalah, server-side discovery lebih sederhana untuk diimplementasikan. Jika biaya menjadi pertimbangan, client-side discovery lebih hemat meskipun lebih kompleks.

- **Service Registry**: Video juga menyebutkan akan membahas service registry di episode berikutnya, yang dapat membantu mengoptimalkan penggunaan client-side discovery.

Dengan demikian, video ini memberikan pemahaman tentang client-side discovery sebagai solusi alternatif yang lebih hemat biaya namun memerlukan pengelolaan lebih kompleks dibandingkan server-side discovery.