Arsitektur Perangkat Lunak

Departemen Informatika, Universitas Pradita, Indonesia

2023

Daftar Isi

Bab 1

Pendahuluan

Alfa Yohannis

1.1 Materi

1. Introduction
2. Client-Server Architecture
3. Model-View-Controller Architecture
4. Model-View-ViewModel Architecture
5. Layered Architecture
6. Event-Driven Architecture
7. Pipeline / Pipe-and-Filter Architecture
8. Service-based (Serverless) Architecture
9. Microkernel Architecture
10. Space-based Architecture
11. Orchestration-driven Service-oriented Architecture
12. Microservices Architecture
13. Containers
14. DevOps

AAAA [**?**]

Bab 2

Arsitektur Client-Server

Alfa Yohannis

2.1 Latar Belakang

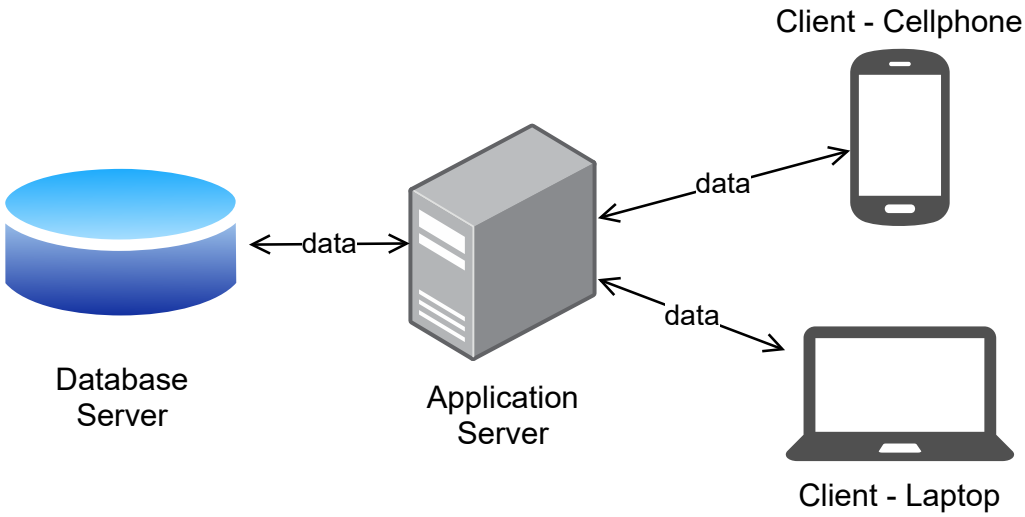
Pada awal komputer bermula sebagai suatu kesatuan, tidak terpisah-pisah. Perangkat lunak hanya berjalan pada satu unit komputer tersebut. Secara perlahan, ada bagian komputer yang dapat terpisah secara fisik dan menjalankan tanggung jawab tertentu. Sebagai contoh, data storage terpisah dari komputer utama. Lalu, beberapa fungsionalitas akhirnya terpisah dan membutuhkan mesin tersendiri. Misalnya, komputer yang didedikasikan untuk menyimpan data atau yang kita sebut sebagai *database server*. Di sisi lain, jaringan komputer juga berkembang dan kemudian menjadi sesautu yang umum. Komputer-komputer saling berkomunikasi satu sama yang lain, dan setiap komputer dapat memiliki peran-peran tertentu yang memungkinkan lahirnya sistem terdistribusi.

2.2 Arsitektur Client-Server

Suatu sistem *client-server* terdiri dari satu *server* dan satu *client* atau lebih. *Server* biasanya memiliki kemampuan komputasi dan penyimpanan data yang lebih cepat dan banyak dibanding *client*. Oleh karena itu, *client* menugaskan *server* untuk melakukan komputasi tertentu dan menerima hasilnya atau sekedar menarik data dari *server*.

Terdapat 2 jenis *client-server architecture*: *two-tier architecture* dan *three-tier architecture*. Two tier-architecture umumnya hanya terdiri dari *desktop application* yang berada di sisi klien dan *database* yang berada di sisi server. Contoh lain adalah *web browser* yang memuat *web application* dan *web server*

untuk melakukan *backend computation*. Arsitektur tersebut dapat diperluas menjadi *three-tier architecture*, dengan menambahkan *database server* seperti yang ditampilkan pada Gambar 14.2.



Gambar 2.1: Skema dari 3-tier client-server arsitektur.

2.3 Kelebihan dan Kekurangan

Berikut adalah kelebihan dan kekurangan arsitektur client-server:

2.3.1 Kelebihan

Keuntungan dari menerapkan arsitektur client-server adalah:

- Kemampuan komputasi (dan penyimpanan data) dapat diakses dari berbagai lokasi berjauhan dan oleh banyak komputer/pengguna.
- Komputasi-komputasi yang membutuhkan kinerja tinggi dapat didelegasikan ke server.
- Data dapat disentralisasikan sehingga meningkatkan konsistensi data dan mengurangi duplikasi data.
- Sistem dapat menerapkan *horizontal scaling* untuk skalabilitas. Horizontal scaling adalah meningkatkan kinerja komputer dengan penambahan komputer agar beban komputasi dibagi ke komputer-komputer

yang tersedia. Misalnya, awalnya terdapat 10 000 requests perhari yang ditangani oleh suatu *application server*. Jika *application server* ditambah, maka beban tersebut dibagi di antara kedua *server* tersebut. Vertical scaling adalah meningkat kinerja suatu komputer dengan menaikkan spefikasi komputer tersebut, misalnya dengan menggunakan prosesor yang lebih cepat atau meningkatkan kapasitas memori.

2.3.2 Kekurangan

Konsekuensi dari penerapan arsitektur client-server adalah sistem jadi lebih kompleks untuk dikelola:

- Biaya akan meningkat karena terdapat komponen/mesin tambahan yang perlu dikelola.
- Faktor keamanan juga perlu diperhatikan karena server dan client beroperasi dalam suatu jaringan komputer yang mana rawan terhadap *cyber attack*.
- Perlunya koordinasi antar-komputer, misalnya komunikasi sinkron dan asinkron serta komputasi parallel.
- Kompatibilitas antara *server* dan *client* maupun sesama klien.
- Masalah-masalah yang umum terdapat pada jaringan komputer etwork problems, misalnya *network latency*, kesalahan dalam konfigurasi jaringan, dsb.

2.4 Contoh Kasus

2.4.1 Deskripsi

Jelaskan contoh kasus yang dipaparkan berkaitan dengan arsitektur yang dimaksud pada bab ini. Contoh kasus harus memperjelas arsitektur yang dimaksud.

2.4.2 Penjelasan Implementasi

Jelaskan bagian-bagian kode program, basisdata, atau konfigurasi yang signifikan terhadap arsitektur yang dimaksud.

2.5 Kesimpulan

Rangkum dan ulangi (beri penekanan pada) hal-hal kunci dari arsitektur yang dimaksud.

Bab 3

Arsitektur MVC (Model-View-Controller)

Alfa Yohannis

3.1 Latar Belakang

Pada mulanya pengembangan perangkat lunak menyatukan fungsi-fungsi dari *graphical user interface* (GUI) dan pengelolaan data ke dalam satu kode tanpa memisahkan mereka sesuai dengan perhatian (*concerns*) mereka masing-masing. Konsekuensinya, pola tersebut akan menimbulkan masalah ketika *developer* diminta untuk membangun aplikasi skala besar, misalnya aplikasi yang menolong pengguna berinteraksi dengan dataset yang besar dan kompleks. Kode program akan menjadi lebih tidak terstruktur (*spaghetti code*) dan sulit untuk dipahami. Sebagai solusi, kode program perlu dibagi ke dalam komponen-komponen sesuai dengan perhatian mereka (*separation of concerns*). Arsitektur Model-View-Controller (MVC) kemudian diajukan untuk membagi kode program ke dalam tiga abstraksi utama: *model*, *view*, dan *controller*.

3.2 Arsitektur Model-View-Controller

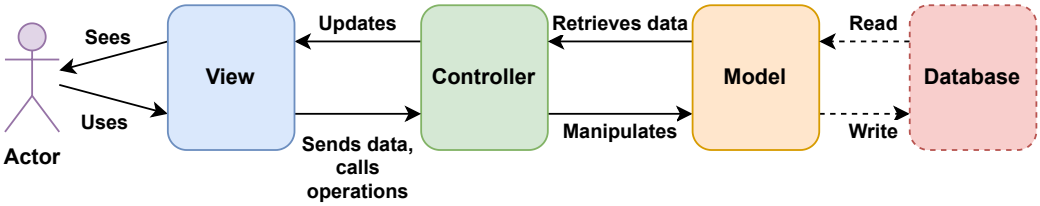
Arsitektur MVC adalah pola arsitektur untuk pengembangan *Graphical User Interface* (GUI). Arsitektur tersebut membagi logika progam menjadi 3 bagian yang saling terhubung: Model, View, dan Controller. Skema dari MVC dapat dilihat pada Gambar 3.1..

Model ditujukan untuk berinteraksi dengan data: menyimpan, memperharui, menghapus, dan menarik data dari database. Model juga digunakan

untuk menggagregasi data sesuai dengan logika bisnis yang dijalankan.

View merupakan presentasi yang ditampilkan ke pengguna yang dengannya pengguna dapat berinteraksi. Misalnya, halaman web, GUI desktop, diagram, *text fields*, *buttons*, dsb.

Controller bertugas untuk menerima input dari pegguna melalui *view* dan meneruskan input tersebut ke model untuk disimpan atau diproses lebih lanjut. Controller juga menarik data dari *model* dan memembetuknya demikian rupa sehingga siap untuk dikirimkan ke *view* untuk ditampilkan ke pengguna.



Gambar 3.1: Arsitektur Model-View-Controller (MVC).

3.3 Kelebihan dan Kekurangan

Berikut adalah kelebihan dan kekurangan arsitektur MVC:

3.3.1 Kelebihan

Keuntungan dari menerapkan arsitektur MVC adalah:

- Pemisahan presentasi dan data membolehkan model ditampilkan di banyak *view* secara bersamaan.
- View bersifat *composable* artinya view dapat dibangun dari berbagai atau berisi *subviews/fragments*.
- Controller satu dapat diganti (*switchable*) dengan controller lain pada saat *runtime*.
- Developer dapat membuat berbagai macam mekanisme pemrosesan data dari input ke output dengan mengkombinasikan berbagai macam fungsionalitas yang dimiliki oleh views, controllers, dan models.
- *Data engineers*, *backend* dan *frontend developers* masing-masing dapat fokus mengerjakan tugas utama mereka. Misal, *data engineers* hanya mengerjakan tugas yang berkaitan dengan data, sendangkan *frontend developers* fokus ke *user interface*.

3.3.2 Kekurangan

Konsekuensi dari penerapan arsitektur MVC adalah sebagai berikut:

- Derajat kompleksitas kode program bertambah karena kode harus dibagi ke dalam tiga abstraksi yang berbeda.
- *Developers* harus mengikuti aturan ketat tertentu dalam mendefinisikan *controllers*, *models*, dan *views*.
- Secara relative, MVC lebih sulit dipahami dikarenakan struktur bawaannya.
- Terlalu berlebihan (*overkill*) untuk aplikasi sederhana.
- Cocok untuk pembangunan Graphical User Interface tetapi belum tentu cocok untuk pengembangan aplikasi atau komponen yang lain.
- Adanya lapisan-lapisan abstraksi dapat mengurangi kinerja (*performance*) aplikasi.

3.4 Contoh Kasus

3.4.1 Deskripsi

Jelaskan contoh kasus yang dipaparkan berkaitan dengan arsitektur yang dimaksud pada bab ini. Contoh kasus harus memperjelas arsitektur yang dimaksud.

3.4.2 Penjelasan Implementasi

Jelaskan bagian-bagian kode program, basisdata, atau konfigurasi yang signifikan terhadap arsitektur yang dimaksud.

Listing 3.1: Model dari Rate.

```
1  import javax.persistence.Entity;
2  import javax.persistence.Id;
3  import javax.persistence.IdClass;
4
5  @Entity
6  @IdClass(RateId.class)
7  public class Rate {
8    @Id
9    private String fromCurrency;
```

```
10    @Id
11    private String toCurrency;
12    private Double rate;
13    ...
14    Rate(String fromCurrency, String toCurrency, Double
          rate) {
15      ...
16    }
17    ...
18  }
```

Listing 3.2: RateRepository.

```
1  import java.util.Collection;
2  import org.springframework.data.jpa.repository.Query;
3  import org.springframework.data.repository.
       CrudRepository;
4
5  public interface RateRepository extends CrudRepository<
       Rate, Integer> {
6    @Query("SELECT r FROM Rate r WHERE r.fromCurrency = ?1
         and r.toCurrency = ?2")
7    Collection<Rate> findFirstByFromCurrencyAndToCurrency(
         String fromCurrency, String toCurrency);
8
9    @Query("SELECT DISTINCT(r.fromCurrency) FROM Rate r")
10   Collection<String> findAllFromCurrency();
11
12   @Query("SELECT DISTINCT(r.toCurrency) FROM Rate r")
13   Collection<String> findAllToCurrency();
14 }
```

3.5 Kesimpulan

Rangkum dan ulangi (beri penekanan pada) hal-hal kunci dari arsitektur yang dimaksud.

Bab 4

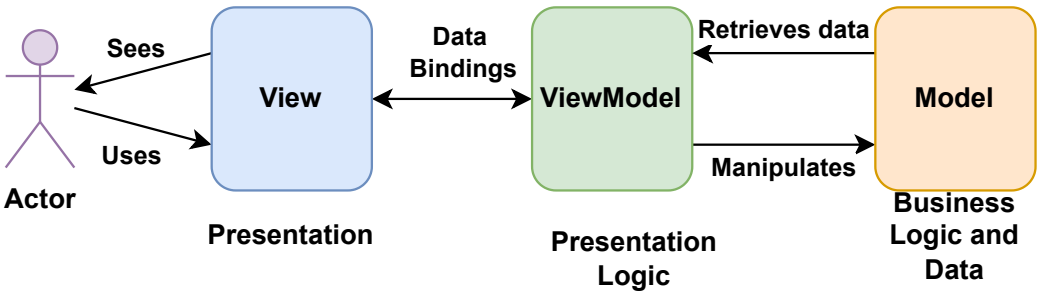
Arsitektur MVVM (Model-View-ViewModel)

Alfa Yohannis

4.1 Latar Belakang

Pada mulanya, dalam pengembangan perangkat lunak, kode yang bertanggung jawab terhadap data, logika bisnis, dan tampilan (Graphical User Interface) bercampur jadi satu, tidak ada pemisahan abstraksi. Pola Model-View-Controller kemudian muncul memisahkan kode program ke dalam 3 abstraksi utama berdasarkan perhatian mereka: *model* untuk data, *view* untuk tampilan, dan *controller* untuk logika bisnis. Hanya saja, MVC tidak memiliki abstraksi yang secara eksplisit mengelola *states* dari tampilan (*views*). Pola MVP (Model-View-Presenter) kemudian diajukan di mana komponen *Presenter*-nya bertanggung jawab mengelola logika presentasi dari *views*. Walaupun demikian,kode program yang mengelola sinkronisasi antara views dan state dari logika presentasi mereka masih harus dibuat secara manual.

Keunikan dari Model-View-ViewModel adalah pola tersebut memiliki komponen *binder* yang mengotomasi komunikasi/sinkronisasi antara view dengan properties yang ada pada *view model*. Nilai-nilai pada *view* ditautkan dengan properties pada view model sehingga perubahan nilai pada salah komponen di view (misalnya perubahan pada *textbox*) akan memperbarui juga nilai pada *property*-nya di *view model* yang ditautkan pada komponen tersebut. Adanya binder mengurangi jumlah kode yang harus ditulis oleh developer secara manual untuk melakukan sinkronisasi antara *view* dan *view model*.



Gambar 4.1: Arsitektur Model-View-ViewModel (MVVM).

4.2 Arsitektur Model-View-ViewModel

- Separation of the view layer by moving all GUI code to the view model via data binding.
- UI developers don't write the the GUI, instead a markup language is used.
- The separation of roles allows UI designers to focus on the UX design rather than programming of the business logic.
- A proper separation of the view from the model is more productive, as the user interface typically changes frequently and late in the development cycle based on end-user feedback.
- Data bindings and properties are used to synchronise the relevant values in the view and the view model, that represents the state of the view, so that they are always the same.
- It eliminates or minimises application logic that directly manipulates the view.

4.3 Kelebihan dan Kekurangan

Berikut adalah kelebihan dan kekurangan arsitektur MVVM:

4.3.1 Kelebihan

Keuntungan dari menerapkan arsitektur MVVM adalah:

- Separation of the view layer by moving all GUI code to the view model via data binding.

- UI developers don't write the the GUI, instead a markup language is used.

- The separation of roles allows UI designers to focus on the UX design rather than programming of the business logic.

- A proper separation of the view from the model is more productive, as the user interface typically changes frequently and late in the development cycle based on end-user feedback.

- Data bindings and properties are used to synchronise the relevant values in the view and the view model, that represents the state of the view, so that they are always the same.

- It eliminates or minimises application logic that directly manipulates the view.

4.3.2 Kekurangan

Konsekuensi dari penerapan arsitektur MVVM adalah sebagai berikut:

- It can be overkill for small projects.

- Generalizing the viewmodel upfront can be difficult for large applications.

- Large-scale data binding can lead to lower performance.

- It's best for UI development but might not the best for other types of developments and applications.

4.4 Contoh Kasus

4.4.1 Deskripsi

Jelaskan contoh kasus yang dipaparkan berkaitan dengan arsitektur yang dimaksud pada bab ini. Contoh kasus harus memperjelas arsitektur yang dimaksud.

4.4.2 Penjelasan Implementasi

Jelaskan bagian-bagian kode program, basisdata, atau konfigurasi yang signifikan terhadap arsitektur yang dimaksud.

Listing 4.1: Model dari Rate.

```
1  import javax.persistence.Entity;
2  import javax.persistence.Id;
3  import javax.persistence.IdClass;
4
5  @Entity
6  @IdClass(RateId.class)
7  public class Rate {
8    @Id
9    private String fromCurrency;
10   @Id
11   private String toCurrency;
12   private Double rate;
13   ...
14   Rate(String fromCurrency, String toCurrency, Double
       rate) {
15     ...
16   }
17   ...
18 }
```

Listing 4.2: RateRepository.

```
1  import java.util.Collection;
2  import org.springframework.data.jpa.repository.Query;
3  import org.springframework.data.repository.
     CrudRepository;
4
5  public interface RateRepository extends CrudRepository<
     Rate, Integer> {
6    @Query("SELECT r FROM Rate r WHERE r.fromCurrency = ?1
       and r.toCurrency = ?2")
7    Collection<Rate> findFirstByFromCurrencyAndToCurrency(
       String fromCurrency, String toCurrency);
8
9    @Query("SELECT DISTINCT(r.fromCurrency) FROM Rate r")
10   Collection<String> findAllFromCurrency();
11
12   @Query("SELECT DISTINCT(r.toCurrency) FROM Rate r")
13   Collection<String> findAllToCurrency();
```

```
14 }
```

4.5 Kesimpulan

Rangkum dan ulangi (beri penekanan pada) hal-hal kunci dari arsitektur yang dimaksud.

Bab 5

Layered Architecture

Austin Nicholas Tham, Darren Valentio, Muhammad

5.1 Definisi *Layered Architechture*

Pola arsitektur *layered* adalah pola *n-tiered* di mana komponen disusun dalam lapisan horizontal. Ini adalah metode tradisional untuk merancang sebagian besar perangkat lunak dan dimaksudkan untuk pengembangan mandiri sehingga semua komponen saling berhubungan tetapi tidak saling bergantung.

Gambar 5.1 *Layering Architecture*

Seperti yang ditunjukkan pada gambar, *layering* biasanya dilakukan dengan mengemas fungsionalitas khusus aplikasi di lapisan atas, penyebaran fungsionalitas spesifik menjadi lapisan bawah dan fungsionalitas yang membentang di seluruh domain aplikasi di lapisan tengah. Jumlah lapisan dan

bagaimana lapisan-lapisan ini disusun ditentukan oleh kompleksitas masalah dan solusinya.

Di sebagian besar arsitektur berlapis, ada beberapa lapisan (atas ke bawah):

- ***The application layered*:** Berisi layanan spesifik aplikasi.
- **The business layer:** Menangkap komponen yang umum di beberapa aplikasi.
- ***The middleware layer*:** Lapisan ini mengemas beberapa fungsi seperti pembangun GUI, antarmuka ke basis data, laporan, dan dll.
- ***The database/System Software Layer*:** Berisi OS, *database*, dan antarmuka ke komponen perangkat keras tertentu.

5.2 Latar Belakang

Penilaian untuk setiap karakteristik berdasarkan kecenderungan alami untuk implementasi tipikal pola *layered*.

- Kemampuan untuk merespon dengan cepat terhadap lingkungan yang terus berubah. (monolitik)
- Bergantung pada implementasi pola, penyebaran bisa menjadi masalah. Satu perubahan kecil ke komponen dapat memerlukan *redeployment* seluruh aplikasi.
- Pengembang dapat memberikan pengujian singkat untuk menguji aplikasi sebelum klien menggunakannya
- Mudah dikembangkan karena polanya sudah terkenal dan tidak terlalu rumit untuk melakukan implementasinya.

5.3 Pros Cons

5.3.1 Pros

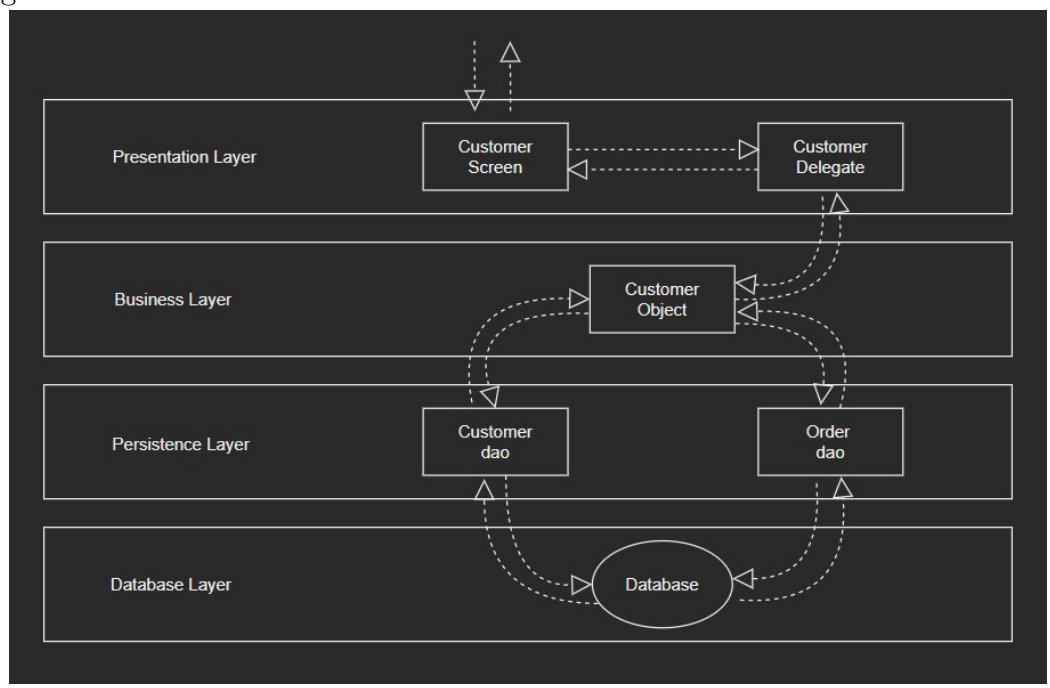
- Mudah untuk diuji karena komponen-komponennya termasuk lapisan khusus sehingga dapat diuji secara terpisah.
- Sederhana dan mudah diimplementasikan karena secara alami, sebagian besar aplikasi bekerja berlapis-lapis

5.3.2 Cons

- Tidak mudah untuk melakukan perubahan pada lapisan tertentu karena aplikasi merupakan unit tunggal.
- Kopling antar lapisan cenderung membuatnya lebih sulit. Hal ini membuatnya sulit untuk diukur.
- Harus digunakan sebagai unit tunggal sehingga perubahan ke lapisan tertentu berarti seluruh sistem harus dipekerjakan kembali.
- Semakin besar, semakin banyak sumber daya yang dibutuhkan untuk permintaan untuk melewati beberapa lapisan dan dengan demikian akan menyebabkan masalah kinerja.

5.4 Software Architechture Pattern

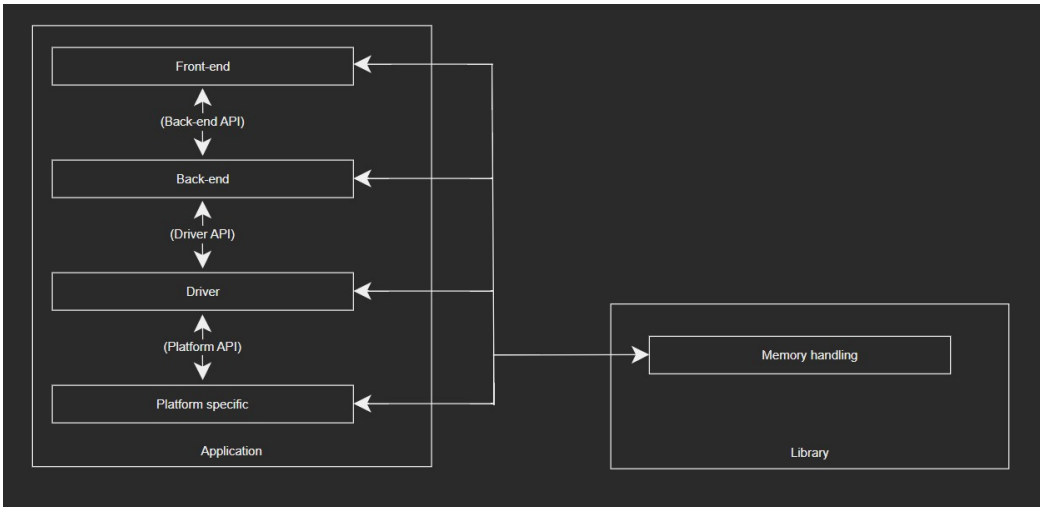
Ini adalah pola arsitektur paling umum di sebagian besar aplikasi tingkat perusahaan. Ini juga dikenal sebagai pola n-tier, dengan asumsi n jumlah tingkatan. Contoh Skenario:



Gambar 5.2 *Software Architecture Pattern*

5.5 Design Patterns

Anggap *mock-up software design*, susunan "*stack*" nya seperti *layered architecture*:



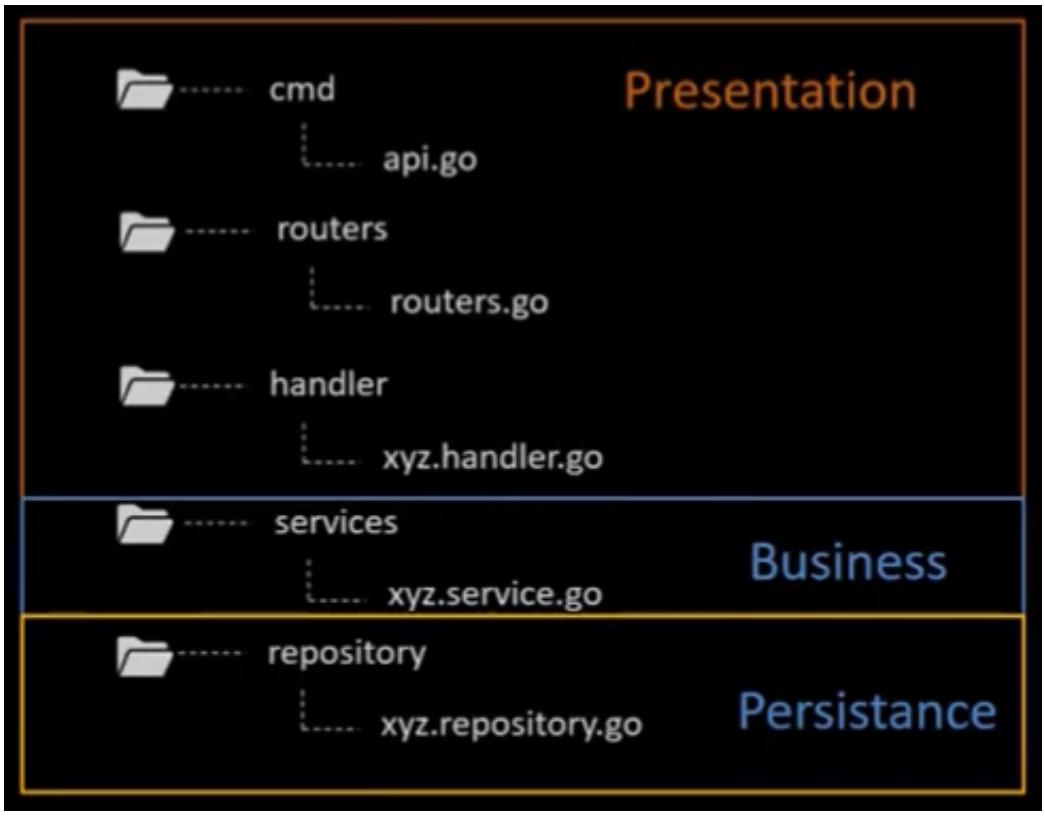
Gambar 5.3 *Design Pattern*

Setiap *layer* dari aplikasi terpisah dengan cara penggunaan metode API, namun yang masih saling berhubungan adalah *memory handling* , karena setiap komunikasi *layer* akan membawa/mengirim data sehingga akan terjadi alokasi *memory* dan pada akhirnya membutuhkan *memory handling*.

Ada 4 bagian dari *layered architecture* yang di mana setiap layer memiliki hubungan antara komponen yang ada di dalamnya dari atas ke bawah yaitu:

- ***The presentation layer*:** Semua bagian yang berhubungan dengan layer presentasi.
- ***The business layer*:** Berhubungan dengan logika bisnis.
- ***The persistence layer*:** Berguna untuk mengurusi semua fungsi yang berhubungan dengan objek relasional
- ***The database layer*:** Tempat penyimpanan semua data layer.

5.5.1 Contoh penerapan *layered architecture*:



Gambar 5.4 Contoh penerapan *layered architecture*

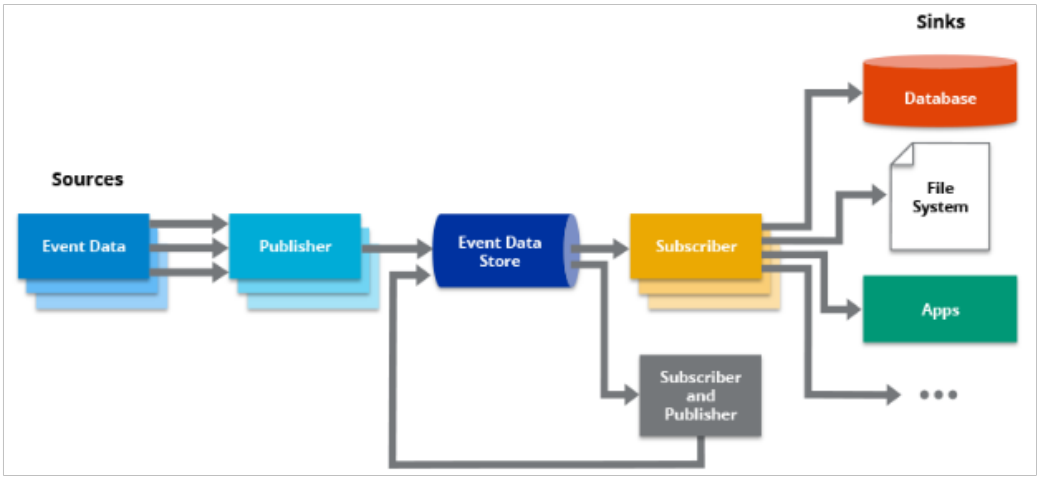
Bab 6

Event-Driven Architecture

Delvin, Gabrielle Sheila Sylvagno, Danica Recca Danendra

6.1 Event-Driven Architecture

6.1.1 Event-Driven Architecture



Gambar 6.1: Skema Diagram EDA

Event-driven architecture (EDA) atau arsitektur berbasis peristiwa adalah paradigma desain perangkat lunak yang memanfaatkan peristiwa *(event)* sebagai dasar interaksi dan integrasi antara komponen-komponen perangkat lunak.EDA berfokus pada peristiwa yang terjadi pada waktu tertentu, seperti

permintaan pengguna atau respons sistem terhadap permintaan tersebut. Komponen-komponen perangkat lunak dalam arsitektur ini saling berkomunikasi melalui peristiwa-peristiwa yang terjadi, sehingga memungkinkan sistem untuk beroperasi secara asinkron.EDA sering digunakan dalam pengembangan aplikasi skala besar dan sistem berbasis layanan *(service-oriented architecture/SOA)* untuk memastikan penggunaan sumber daya yang efektif dan efisien. Arsitektur ini juga dapat membantu meminimalkan waktu respon dan meningkatkan skalabilitas sistem.Berikut ini diagram EDA yang ditampilkan pada 6.1

6.2 Kelebihan dan Kekurangan

6.2.1 Kelebihan

Berikut ini adalah kelebihan dari EDA:

- Asinkron: EDA memungkinkan komponen sistem beroperasi secara asinkron, yaitu mereka dapat beroperasi secara independen tanpa harus menunggu komponen lainnya untuk menyelesaikan tugasnya.
- Pemicu: EDA didasarkan pada penggunaan peristiwa sebagai pemicu untuk memicu tindakan atau respons. Ketika peristiwa terjadi, EDA akan memicu tindakan yang sesuai dengan peristiwa tersebut.
- Publikasi dan Langganan: EDA menggunakan model publikasi-langganan *(publish-subscribe)* dimana sebuah komponen menghasilkan peristiwa *(publisher)* dan komponen lainnya yang tertarik *(subscriber)* dapat menerima dan menangani peristiwa tersebut.
- Terdistribusi: EDA memungkinkan komponen sistem tersebar di berbagai mesin atau jaringan, sehingga memudahkan pengembangan sistem yang *scalable* dan tahan bencana.
- Flesksibel dan modular: EDA memisahkan komponen-komponen sistem sehingga mereka dapat beroperasi secara independen dan dapat digunakan kembali dalam berbagai aplikasi atau sistem yang berbeda.
- Responsif: EDA memungkinkan sistem merespons permintaan dengan cepat, karena komponen sistem dapat beroperasi secara independen dan merespons peristiwa secara asinkron.
- Berorientasi pada pesan: EDA menggunakan pesan sebagai sarana untuk berkomunikasi antar komponen sistem. Pesan dapat mengandung data atau informasi yang diperlukan oleh komponen lain dalam sistem.

- Skalabel: EDA dapat diimplementasikan pada sistem yang memiliki tingkat skala dan kompleksitas yang berbeda-beda, mulai dari sistem skala kecil hingga sistem skala besar dan terdistribusi.

6.2.2 Kekurangan

Berikut ini adalah kekurangan dari EDA:

- Kompleksitas: EDA bisa menjadi sangat kompleks karena banyaknya komponen dan interaksi antar komponen dalam sistem. Hal ini dapat membuat pengembangan dan pemeliharaan sistem menjadi lebih sulit.
- Kesulitan dalam pemantauan dan manajemen: Dalam EDA, setiap peristiwa dapat dicatat dan dilacak, namun hal ini bisa menyebabkan sulitnya pemantauan dan manajemen sistem jika terdapat banyak peristiwa yang terjadi pada waktu yang sama.
- Kemungkinan kesalahan: Karena EDA melibatkan banyak komponen yang berinteraksi satu sama lain, maka kemungkinan terjadinya kesalahan atau bug dalam sistem juga semakin besar. Hal ini dapat menyebabkan kerusakan sistem atau bahkan kegagalan total dalam sistem. /item Tidak cocok untuk sistem yang simpel: EDA biasanya digunakan pada sistem yang kompleks dan memerlukan integrasi dengan berbagai sistem atau aplikasi lainnya. Sehingga EDA mungkin tidak cocok untuk sistem yang simpel atau terbatas dalam kompleksitasnya.

6.3 Contoh Penerapan

6.3.1 Perbankan

Sistem perbankan: EDA dapat digunakan untuk membangun sistem perbankan yang responsif dan skalabel. Contohnya adalah ketika seorang pelanggan melakukan transfer uang, hal ini memicu peristiwa (event) yang kemudian membuat sistem mengirimkan notifikasi kepada penerima transfer bahwa uang telah diterima.

6.3.2 E-commerce

Aplikasi e-commerce: EDA dapat digunakan dalam aplikasi e-commerce untuk mempercepat proses pembelian. Ketika seorang pelanggan menyelesaikan pembelian, peristiwa ini dapat memicu sistem untuk mengirim notifikasi ke bagian pengiriman dan bagian keuangan untuk memproses pesanan.

6.3.3 Internet of Thing (IoT)

Internet of Things (IoT): EDA juga dapat digunakan dalam sistem IoT, di mana banyak sensor dan perangkat harus berinteraksi dengan sistem pusat. Contohnya adalah ketika suhu di suatu ruangan melebihi batas normal, peristiwa ini dapat memicu sistem untuk mengirim notifikasi ke teknisi untuk memperbaiki perangkat pendingin ruangan.

6.3.4 Manajemen Rantai Pasokan

Sistem manajemen rantai pasokan: EDA dapat digunakan dalam sistem manajemen rantai pasokan untuk memantau pergerakan barang dari satu titik ke titik lainnya. Ketika sebuah produk telah dikirim, peristiwa ini dapat memicu sistem untuk mengirim notifikasi ke penerima produk tentang waktu pengiriman yang dijadwalkan.

6.3.5 Manajemen Proyek

Sistem manajemen proyek: EDA dapat digunakan dalam sistem manajemen proyek untuk memantau perkembangan proyek dan memperingatkan manajer proyek ketika terjadi masalah atau penundaan. Contohnya, ketika seorang anggota tim menyelesaikan tugas mereka, peristiwa ini dapat memicu sistem untuk memperbarui proyek secara otomatis dan memberikan notifikasi kepada manajer proyek.

Bab 7

Pendahuluan

Alfa Yohannis, Rizki Wahyudi, Tommy Chitiawan, Mandalan

Gny: Tolong tambahkan keterangan gambar contoh : Gambar 7.1, 7.2, dst.. dan tambahkan italic text untuk setiap bahasa asing

7.1 Definisi

Pipe and Filter Architecture adalah sebuah pendekatan desain perangkat lunak yang menggambarkan bagaimana data dapat diproses melalui serangkaian *filter* atau pemroses yang saling terkait dan saling bergantung dalam suatu *pipeline*. Setiap *filter* memiliki tugas spesifik untuk mengubah atau memanipulasi data yang melewatinya, dan data tersebut kemudian dikirim ke *filter* berikutnya dalam *pipeline* untuk diproses lebih lanjut.

Pipe and Filter Architecture terdiri dari beberapa elemen utama, yaitu:

- *Pipes*: adalah saluran yang menghubungkan antara satu *filter* dengan *filter* lainnya. Pipe digunakan untuk mengalirkan data dari satu *filter* ke *filter* berikutnya.

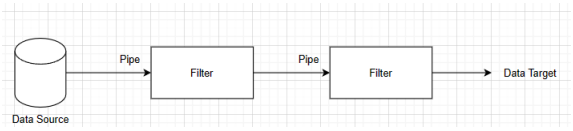
- *filter*: adalah blok bangunan logika yang bertanggung jawab untuk memproses dan mengubah data. *Filter* dapat melakukan tugas sederhana seperti memisahkan atau menyaring data, atau tugas yang lebih kompleks seperti mengubah format data.

- *Source dan Sink*: adalah elemen yang menghasilkan *input* data dan menerima *output* data dari *pipeline*.

Keuntungan utama dari *Pipe and Filter Architecture* adalah bahwa ia memungkinkan pengembang untuk membangun sistem yang sangat modular, dengan setiap *filter* melakukan tugas yang jelas dan terbatas. Hal ini membuat perubahan pada pipeline lebih mudah dan aman, karena hanya memerlukan perubahan pada satu *filter* tanpa mempengaruhi *filter* lainnya. Selain itu, arsitektur ini juga dapat meningkatkan kinerja sistem, karena memungkinkan untuk memproses data secara paralel dalam beberapa *filter*.

Namun, kelemahan dari *Pipe and Filter Architecture* adalah bahwa dapat menjadi sulit untuk menangani kasus penggunaan yang kompleks, karena setiap filter harus dirancang dengan sangat baik agar dapat berjalan dengan benar dalam pipeline. Selain itu, pengembang harus memperhatikan antarmuka antara filter yang berbeda agar dapat saling berinteraksi dengan benar.

7.2 *Pipe and Filter Architecture Schema*



Gambar 7.1: *pipe and filter*

7.3 Kelebihan

- Memastikan sambungan komponen, *filter* yang longgar dan fleksibel.
- Kopling longgar memungkinkan *filter* diubah tanpa modifikasi ke *filter* lain.
- Konduktif untuk pemrosesan paralel.
- *filter* dapat diperlakukan sebagai kotak hitam. Pengguna sistem tidak perlu mengetahui logika di balik kerja setiap *filter*.
- Dapat digunakan kembali. Setiap *filter* dapat dipanggil dan digunakan berulang kali.

7.4 Kekurangan

- Penambahan sejumlah besar *filter independent* dapat mengurangi kinerja karena overhead komputasi yang berlebihan.
- Bukan pilihan yang baik untuk sistem interaktif.
- Sistem *pipe and filter* mungkin tidak cocok untuk perhitungan jangka panjang.

7.5 penerapan dalam aplikasi

- Sistem pengolahan data: *Pipe and filter* dapat digunakan untuk mengambil data dari berbagai sumber dan memprosesnya melalui serangkaian *filter* untuk menghasilkan *output* yang diinginkan.
- Sistem pengolahan gambar: *Pipe and filter* dapat digunakan untuk memproses gambar atau video yang diambil dari kamera dengan menggunakan berbagai filter untuk menghasilkan gambar yang lebih baik atau memberikan efek khusus.
- Sistem pencarian: *Pipe and filter* dapat digunakan untuk memproses data pencarian yang diberikan oleh pengguna dan memfilter data untuk menghasilkan hasil pencarian yang relevan.
- Sistem pemrosesan audio: *Pipe and filter* dapat digunakan untuk memproses audio dan melakukan pengolahan suara seperti pengurangan kebisingan, pengaturan volume, dan pemotongan audio.
- Sistem pemrosesan teks: *Pipe and filter* dapat digunakan untuk memproses teks dan melakukan pengolahan bahasa alami seperti analisis sentimen dan pengenalan entitas.

Bab 8

Serverless Architecture

Ryan Christensen Wang, Steven Tanaka, Yogi Valentino Nadeak

Gny: Tolong tambahkan keterangan gambar contoh : Gambar 8.1, 8.2, dst.. dan tambahkan italic text untuk setiap bahasa asing

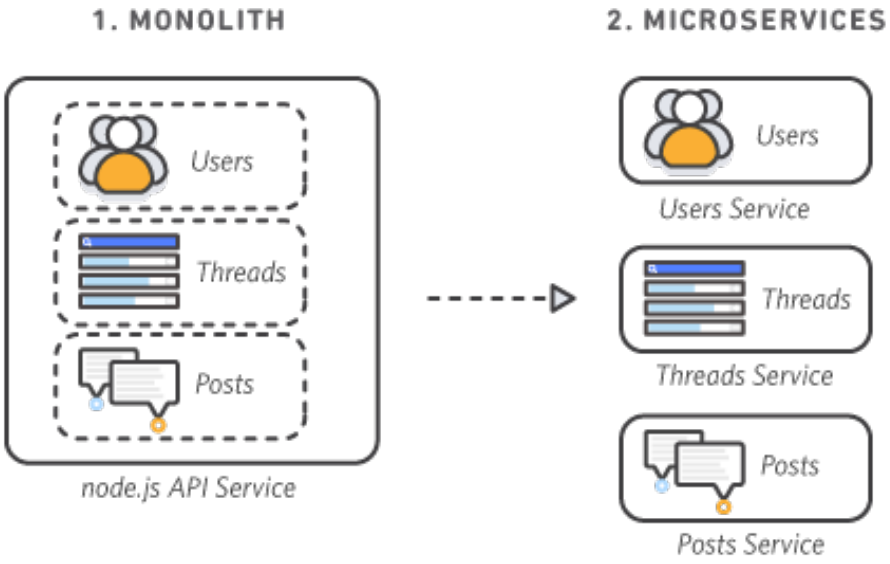
8.1 Definisi Serverless Architechture

Serverless architecture merupakan pendekatan dalam desain software yang mana developer tidak perlu pusing-pusing lagi mengelola infrastruktur seperti server.Developer bisa fokus dalam mengembangkan aplikasinya dan masalah server akan dikelola oleh penyedia layanan serverless (provider).

Less dalam serverless bukan berarti tanpa server sama sekali, tetapi memungkinkan untuk konfigurasi seminimal mungkin atau dikurangi. Misalnya, pengembang tidak perlu tidak perlu otak-atik konfigurasi web server ketika melakukan deploy kode yang hanya tinggal dihubungkan ke proxy.

Developer menggunakan Serverless Architecture untuk menggunakan kembali layanan dalam sistem yang berbeda atau menggabungkan beberapa layanan independen untuk melakukan tugas yang kompleks.

Misalnya, beberapa proses bisnis dalam suatu organisasi memerlukan fungsionalitas autentikasi pengguna. Alih-alih menulis ulang kode autentikasi untuk semua proses bisnis, Anda dapat membuat satu layanan autentikasi dan menggunakannya kembali untuk semua aplikasi. Demikian juga dengan sistem di seluruh organisasi layanan kesehatan, seperti sistem manajemen pasien dan sistem catatan kesehatan elektronik (EHR), sebagian besar perlu mendaftarkan pasien. Sistem ini dapat memanggil satu layanan umum untuk melakukan tugas pendaftaran pasien.

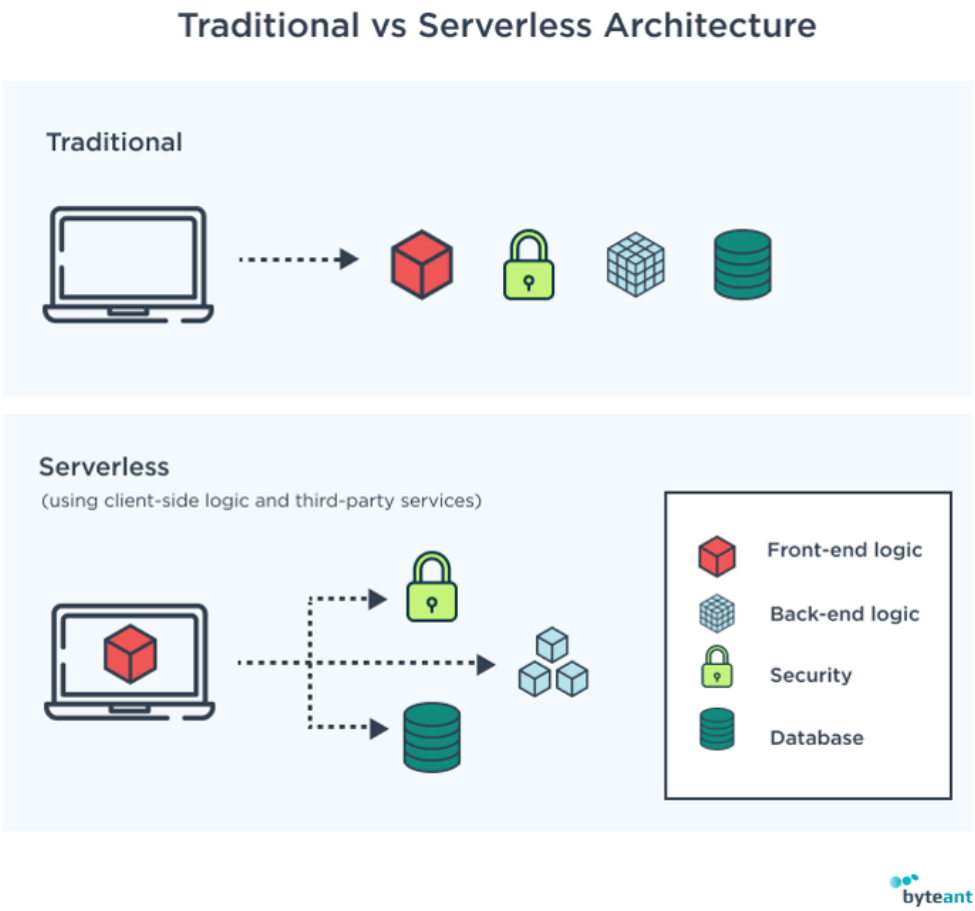


Ada beberapa istilah dasar dalam arsitektur ini, antara lain:

- **Invocation,** yaitu eksekusi fungsi tunggal.
- **Duration,** durasi waktu yang dibutuhkan fungsi serverless
- **Cold start,** yaitu terjadinya latensi saat fungsi ter-trigger pertama kali atau setelah periode tidak aktif.
- **Concurrency limit,** yaitu jumlah fungsi instance yang dapat dijalankan secara bersamaan dalam satu wilayah
- **Timeout,** yaitu jumlah waktu yang diizinkan oleh penyedia layanan untuk menjalankan fungsi sebelum dihentikan.

Serverless architecture ini cocok digunakan untuk kasus yang melakukan tugas jangka pendek dan mengelola beban kerja yang mengalami lalu lintas yang jarang dan tidak dapat diprediksi. Ada beberapa kasus lain yang dapat dipertimbangkan menggunakan arsitektur ini, yaitu:

- Tugas yang berdasarkan trigger
- Membangun RESTful API
- Continuous Integration dan Continuous Delivery (CI/CD)
- Proses asinkronus



8.2 Fungsi/Kegunaan dari Serverless Architecture

Serverless architecture adalah sebuah model komputasi di awan yang mana penyedia layanan awan bertanggung jawab dalam mengelola infrastruktur dan memperuntukan sumber daya komputasi yang dibutuhkan secara otomatis, tanpa pengguna perlu mengurus atau merawat server. Terdapat beberapa keuntungan dari penggunaan arsitektur serverless, diantaranya adalah:

Penilaian untuk setiap karakteristik berdasarkan kecenderungan alami untuk implementasi tipikal pola layered.

- optimisasi biaya dengan hanya membayar untuk sumber daya yang digunakan

- memungkinkan pengembangan aplikasi yang lebih cepat
- skalabilitas yang mudah untuk mengakomodasi perubahan permintaan
- serta perawatan dan pemeliharaan yang lebih mudah karena dikelola oleh penyedia layanan awan.

8.3 Kekurangan dan Kelebihan dari Serverless Architecture

8.3.1 Kelebihan dari Serverless Architecture

- **Skalabilitas:** Dalam arsitektur serverless, aplikasi dapat dengan mudah ditingkatkan kapasitasnya untuk menangani permintaan yang berfluktuasi, sehingga dapat mengurangi pengeluaran untuk infrastruktur yang tidak terpakai.
- **Biaya operasional yang rendah:** Dalam serverless, pengguna hanya membayar untuk sumber daya yang mereka gunakan, yang dapat mengurangi biaya operasional secara signifikan.
- **Fokus pada pengembangan aplikasi:** Dalam serverless, pengembang tidak perlu khawatir mengurus infrastruktur server dan dapat fokus pada pengembangan aplikasi.
- **Perawatan dan pemeliharaan yang mudah:** Dalam serverless, penyedia layanan awan bertanggung jawab atas perawatan dan pemeliharaan infrastruktur, sehingga pengguna tidak perlu memikirkan pembaruan sistem operasi atau patch keamanan.

8.3.2 Kekurangan dari Serverless Architecture

- **Keterbatasan dalam penggunaan:** Serverless mungkin tidak cocok untuk semua jenis aplikasi, terutama jika aplikasi memerlukan kontrol tinggi atas infrastruktur dan lingkungan di mana aplikasi berjalan.
- **Ketergantungan pada penyedia layanan awan:** Serverless membuat pengguna sangat bergantung pada penyedia layanan awan, sehingga jika terjadi masalah atau gangguan pada layanan, aplikasi dapat mengalami downtime yang signifikan.

- **Pengaturan konfigurasi yang kompleks:** Serverless dapat memiliki konfigurasi yang kompleks dan memerlukan pengaturan yang cermat untuk memastikan aplikasi berjalan dengan baik.

- **Performa yang tidak stabil:** : Serverless dapat mengalami performa yang tidak stabil jika pengguna tidak melakukan penyesuaian yang cermat dalam skala dan konfigurasi aplikasi.

8.4 Penerapan Serverless Architecture

Serverless Architecture adalah model komputasi awan di mana penyedia awan secara dinamis mengelola alokasi dan penyediaan server, memungkinkan pengembang untuk fokus menulis kode tanpa harus mengelola infrastruktur.

Berikut adalah beberapa contoh penerapan serverless architecture:

- **Web applications:** Pengembang dapat membangun dan menerapkan aplikasi web menggunakan arsitektur tanpa server, tanpa harus mengelola server atau infrastruktur. Mereka dapat menggunakan layanan seperti AWS Lambda, Google Cloud Functions, atau Azure Functions untuk menulis kode yang merespons kejadian, seperti permintaan HTTP, dan berjalan sesuai permintaan.

- **Data processing:** Serverless Architecture dapat digunakan untuk tugas pemrosesan data seperti transformasi data, pembersihan, dan analisis. Pengembang dapat menggunakan layanan seperti AWS Glue, Google Cloud Dataflow, atau Azure Stream Analytics untuk memproses data tanpa server, tanpa harus mengelola server atau infrastruktur.

- **Chatbots:** Pengembang dapat membangun chatbot menggunakan serverless, dengan menulis kode yang merespons acara obrolan, seperti pesan pengguna. Mereka dapat menggunakan layanan seperti AWS Lex, Google Cloud Dialogflow, atau Azure Bot Service untuk membuat dan menerapkan chatbot yang berjalan sesuai permintaan.

- **IoT applications:** Serverless Architecture dapat digunakan untuk aplikasi IoT, dengan memungkinkan pengembang menulis kode yang merespons kejadian dari perangkat IoT, seperti pembacaan sensor. Mereka dapat menggunakan layanan seperti AWS IoT, Google Cloud IoT Core, atau Azure IoT Hub untuk membangun dan menerapkan aplikasi IoT tanpa server.

Bab 9

Pendahuluan

Alfa Yohannis, Charlie Chaplin

9.1 Materi

1. Introduction
2. Client-Server Architecture
3. Monolith vs. Distributed Architecture
4. Model-View-Controller Architecture
5. Layered Architecture
6. Event-Driven Architecture
7. Pipeline / Pipe-and-Filter Architecture
8. Service-based (Serverless) Architecture
9. Microkernel Architecture
10. Space-based Architecture
11. Orchestration-driven Service-oriented Architecture
12. Microservices Architecture
13. Containers
14. DevOps

Bab 10

Space-Based Architectur

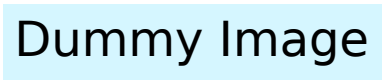
David Eri Nugroho

Latar Belakang

Asal mula terciptanya Space-Based Architecture ini dikarenakan adanya kondisi Triangle-Shaped Topology, yaitu suatu kondisi ketika kita melakukan scalability dengan cara menambah jumlah aplikasi, server, API, database, ataupun aplikasi lain untuk mengatasi kelambatan di sistem kita.

Biasanya kelambatan ini terjadi karena adanya jumlah traffic visitor yang tidak terduga, yang itu berarti traffic visitor dapat membludak sewaktu-waktu, misalnya seperti situs e-commerce, aplikasi penjualan tiket baik app maupun web, serta game.

Dalam kasus ini, jumlah web server misalnya, bisa jauh lebih banyak daripada API, dan jumlah API lebih banyak daripada



Gambar 10.1: Triangle-Shape Topology

Dummy Image

Gambar 10.2: Space-Based Architecture Versi 1

jumlah Database. Hal ini tidak masalah jika jumlah pengunjung masih dalam batas kendali sistem. Jika sistem sudah tidak bisa menampung jumlah pengunjung, maka harus dilakukan scalling dengan menambah jumlah server, API, maupun database.

Namun, melakukan scalling tentu tidak bisa dilakukan dengan mudah. Perlu banyak proses untuk melakukan scalling, seperti replikasi dan proses lainnya. Hal ini mirip seperti yang terjadi pada kasus gerbang tol, yang mana para pengguna jalan harus melewati gerbang tol untuk dapat melanjutkan perjalanan.

Kadangkala hal seperti itulah yang menyebabkan kemacetan. Ini juga terjadi dalam kasus ini. Oleh karena itu dibuatlah suatu arsitektur aplikasi yang dapat mengatasi masalah ini, sehingga penggunaan aplikasi bisa menjadi lebih optimal, yaitu Space-Based Architecture.

Pengertian

Space-Based Architecture adalah pendekatan untuk sistem komputasi terdistribusi di mana berbagai komponen berinteraksi satu sama lain dengan bertukar tupel atau entri melalui satu atau lebih ruang bersama. Hal ini berlawanan dengan pendekatan layanan message queueing yang lebih umum di mana berbagai komponen berinteraksi satu sama lain dengan bertukar pesan melalui message broker.

Space-Based Architecture (terkadang disebut sebagai cloud architecture pattern atau pola arsitektur awan) dirancang khusus

untuk mengatasi dan memecahkan masalah skalabilitas yang ekstrem dan konkurensi. Pola ini juga berguna untuk aplikasi yang volume penggunanya tidak dapat diprediksi. Pola ini dinamakan berdasar pada konsep tuple space dimana menggunakan shared memory yang terdistribusi.

Space-based architecture (SBA) adalah arsitektur perangkat lunak yang didesain untuk mengatasi masalah skalabilitas dan kinerja yang kompleks pada sistem distribusi. SBA didasarkan pada konsep space, yaitu kumpulan data yang dikelompokkan berdasarkan konteks dan dibagi ke dalam cluster-cluster yang terdistribusi secara geografis.

SBA memungkinkan aplikasi untuk memproses data secara parallel dan terdistribusi pada beberapa node atau server yang terhubung, sehingga meningkatkan kinerja dan skalabilitas sistem. SBA juga memungkinkan aplikasi untuk memproses data secara real-time dan memberikan respons yang cepat terhadap permintaan pengguna.

SBA menggunakan beberapa teknologi seperti middleware message-oriented, data grid, dan virtualization untuk membangun sistem yang terdistribusi dan terintegrasi dengan baik. Beberapa contoh teknologi yang digunakan dalam SBA antara lain Apache Kafka, Apache Ignite, dan Docker.

SBA dapat digunakan dalam berbagai jenis aplikasi seperti e-commerce, manufaktur, telekomunikasi, dan lain-lain. SBA sangat cocok untuk aplikasi yang membutuhkan skalabilitas dan kinerja yang tinggi, seperti aplikasi e-commerce yang memproses ribuan transaksi per detik atau aplikasi telekomunikasi yang memproses jutaan panggilan dan pesan per hari.

Berikut ini adalah contoh kerja space-based architecture (SBA) pada e-commerce:

1. Memisahkan data transaksi dari aplikasi e-commerce utama dan menyimpannya di dalam data grid yang terdistribusi. Data grid dapat diimplementasikan menggunakan teknologi seperti Apache Ignite atau Hazelcast.

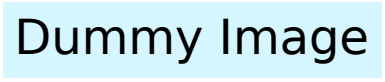
2. Menentukan konteks dan kunci unik untuk setiap transaksi, sehingga memungkinkan pengelompokan dan akses data secara efisien. Konteks dapat berupa informasi pembayaran, informasi pengiriman, atau informasi produk. Kunci unik dapat berupa nomor pesanan atau nomor transaksi.

3. Memperbarui atau mengakses data transaksi dengan mengirimkan permintaan ke data grid. Permintaan tersebut dapat berupa operasi CRUD (Create, Read, Update, Delete) atau operasi lain yang sesuai dengan kebutuhan aplikasi e-commerce.

4. Menggunakan middleware message-oriented seperti Apache Kafka atau RabbitMQ untuk mengintegrasikan aplikasi e-commerce dengan sistem lain. Middleware message-oriented memungkinkan aplikasi untuk melakukan pub/sub model dan mengirimkan pesan antar komponen atau server.

5. Menggunakan teknologi virtualisasi seperti Docker atau Kubernetes untuk mengelola dan mengontrol kontainer aplikasi yang terdistribusi. Teknologi virtualisasi memungkinkan aplikasi untuk berjalan pada lingkungan yang terisolasi dan terpisah, sehingga meningkatkan keamanan dan stabilitas sistem.

6. Menggunakan teknologi monitoring dan logging seperti Prometheus atau ELK stack untuk memonitor kinerja dan performa sistem. Monitoring dan logging memungkinkan aplikasi untuk



Gambar 10.3: Space-Based Architecture Versi 2

mendeteksi dan memperbaiki masalah sebelum mempengaruhi pengguna.

Sejarah

Space-based architecture (SBA) awalnya ditemukan dan dikembangkan di Microsoft pada tahun 1997–98. Secara internal di Microsoft dikenal sebagai Youkon Distributed Caching platform (YDC). Proyek web besar pertama berdasarkan itu adalah MSN Live Search (dirilis pada September 1999) dan kemudian penyimpanan data pemasaran Pelanggan MSN (DB dalam memori multi-terabyte yang dibagikan oleh semua situs MSN) serta sejumlah situs MSN lainnya yang dirilis pada akhir 1990-an dan awal 2000-an.

Jenis-jenis Space-Based Achitecture

Tuple Space

- setiap ruang seperti 'saluran' dalam sistem perantara pesan yang dapat dipilih oleh komponen untuk berinteraksi
- komponen dapat menulis 'tuple' atau 'entry' ke dalam spasi, sementara komponen lain dapat membaca entri/tuple dari space, tetapi menggunakan mekanisme yang lebih kuat daripada perantara pesan

- menulis entri ke spasi umumnya tidak dipesan seperti pada broker pesan, tetapi bisa jika perlu
- merancang aplikasi menggunakan pendekatan ini kurang intuitif bagi kebanyakan orang, dan dapat menghadirkan lebih banyak muatan kognitif untuk diapresiasi dan dieksploitasi

Ruang tuple adalah implementasi dari paradigma memori asosiatif untuk komputasi paralel/terdistribusi. Ini menyediakan repositori tupel yang dapat diakses secara bersamaan.

Message Broker

- setiap broker biasanya mendukung beberapa 'saluran' yang dapat dipilih oleh komponen untuk berinteraksi
- komponen menulis 'pesan' ke saluran, sementara komponen lain membaca pesan dari saluran
- menulis pesan ke saluran umumnya berurutan, di mana pesan umumnya dibaca dalam urutan yang sama
- merancang aplikasi menggunakan pendekatan ini lebih intuitif bagi kebanyakan orang, seperti database NoSQL lebih intuitif daripada SQL

Data Grid

Data Grid mungkin merupakan komponen yang paling penting dan krusial dalam pola ini. Kisi data berinteraksi dengan mesin replikasi data di setiap unit pemrosesan untuk mengelola replikasi data antar unit pemrosesan saat pembaruan data terjadi. Karena kotak perpesanan dapat meneruskan permintaan ke salah satu unit pemroses yang tersedia, setiap unit pemroses

harus berisi data yang persis sama dalam kisi data dalam memorinya.

Messaging Grid

Messaging Grid mengelola permintaan masukan dan informasi sesi. Saat permintaan masuk ke komponen middleware tervirtualisasi, komponen jaringan pesan menentukan komponen pemrosesan aktif mana yang tersedia untuk menerima permintaan dan meneruskan permintaan ke salah satu unit pemrosesan tersebut. Kompleksitas kotak perpesanan dapat berkisar dari algoritme round-robin sederhana hingga algoritme next-available yang lebih kompleks yang melacak permintaan mana yang sedang diproses oleh unit pemrosesan mana.

Processing Grid

Processing Grid adalah komponen opsional dalam middleware tervirtualisasi yang mengelola pemrosesan permintaan terdistribusi ketika terdapat beberapa unit pemrosesan, masing-masing menangani sebagian dari aplikasi. Jika permintaan masuk yang memerlukan koordinasi antara jenis unit pemrosesan (misalnya, unit pemrosesan pesanan dan unit pemrosesan pelanggan), jaringan pemrosesanlah yang memediasi dan mengatur permintaan antara dua unit pemrosesan tersebut.

Deployment Manager

Deployment Manager mengelola startup dinamis dan shutdown unit pemrosesan berdasarkan kondisi beban. Komponen ini terus memantau waktu respons dan beban pengguna, dan memulai unit pemrosesan baru saat beban meningkat, dan mematikan unit pemrosesan saat beban berkurang. Ini adalah komponen

penting untuk mencapai kebutuhan skalabilitas variabel dalam aplikasi.

Kelebihan dan Kekurangan

Kelebihan

- Merespons dengan cepat terhadap lingkungan yang terus berubah.
- Meskipun arsitektur berbasis ruang umumnya tidak dipisahkan dan didistribusikan, mereka dinamis, dan alat berbasis cloud yang canggih memungkinkan aplikasi untuk dengan mudah "didorong" ke server, menyederhanakan penerapan.
- Performa tinggi dicapai melalui akses data dalam memori dan mekanisme caching yang dibangun ke dalam pola ini.
- Skalabilitas tinggi berasal dari fakta bahwa ada sedikit atau tidak ada ketergantungan pada database terpusat, oleh karena itu pada dasarnya menghilangkan hambatan yang membatasi ini dari persamaan skalabilitas.

Kekurangan

- Mencapai beban pengguna yang sangat tinggi dalam lingkungan pengujian adalah hal yang mahal dan memakan waktu, sehingga sulit untuk menguji aspek skalabilitas aplikasi.
- Caching yang canggih dan produk grid data dalam memori membuat pola ini relatif kompleks untuk dikembangkan, sebagian besar karena kurangnya pemahaman tentang alat

dan produk yang digunakan untuk membuat jenis arsitektur ini. Selain itu, perhatian khusus harus diberikan saat mengembangkan jenis arsitektur ini untuk memastikan tidak ada kode sumber yang memengaruhi kinerja dan skalabilitas.

Implementasi

Space-based architecture adalah sebuah arsitektur perangkat lunak yang terdiri dari beberapa komponen atau node yang bekerja sama secara terdistribusi dan saling terhubung melalui jaringan komunikasi. Beberapa contoh aplikasi space-based architecture adalah:

- **E-commerce:** Sebuah aplikasi e-commerce memerlukan sistem yang mampu menangani banyak transaksi dan permintaan yang berbeda-beda dari pengguna. Space-based architecture dapat digunakan untuk mempercepat kinerja aplikasi e-commerce dengan memperluas kemampuan skalabilitas dan toleransi kesalahan yang lebih besar.
- **Permainan online:** Permainan online memerlukan sistem yang mampu menangani banyak pemain dalam satu waktu dan menyediakan pengalaman yang konsisten untuk setiap pemain. Space-based architecture dapat digunakan untuk memperluas kemampuan game server dalam menangani jumlah pemain yang lebih besar dan memberikan pengalaman game yang lebih responsif.
- **Sistem sensor jaringan:** Sistem sensor jaringan yang digunakan dalam lingkungan yang luas atau di permukaan bumi dapat memanfaatkan space-based architecture untuk memperluas cakupan dan memperkuat kinerja dengan menyediakan jaringan yang lebih terdistribusi dan skalabel.

- **Sistem analisis data:** Sistem analisis data yang memerlukan kinerja yang tinggi dan skalabilitas dapat mengadopsi space-based architecture untuk mempercepat waktu respon dan memperluas kemampuan data processing yang lebih besar.
- **Aplikasi internet of things:** Aplikasi internet of things yang memerlukan konektivitas yang tinggi antara perangkat dan sistem dapat mengadopsi space-based architecture untuk mempercepat respons waktu dan meningkatkan kinerja sistem secara keseluruhan.

Bab 11

Orchestration-driven Service-oriented Architecture

Hansel Ricardo, Jonathan Erik Maruli Tua, Yefta Tanuwijaya

11.1 Definisi

Orchestration-driven Service-oriented Architecture (ODSOA) adalah suatu pendekatan arsitektur perangkat lunak yang bertujuan untuk memfasilitasi pengembangan dan integrasi sistem yang kompleks dengan cara menggunakan layanan (*Services*) yang terdistribusi dan terpisah secara fisik namun saling terkait secara fungsional.

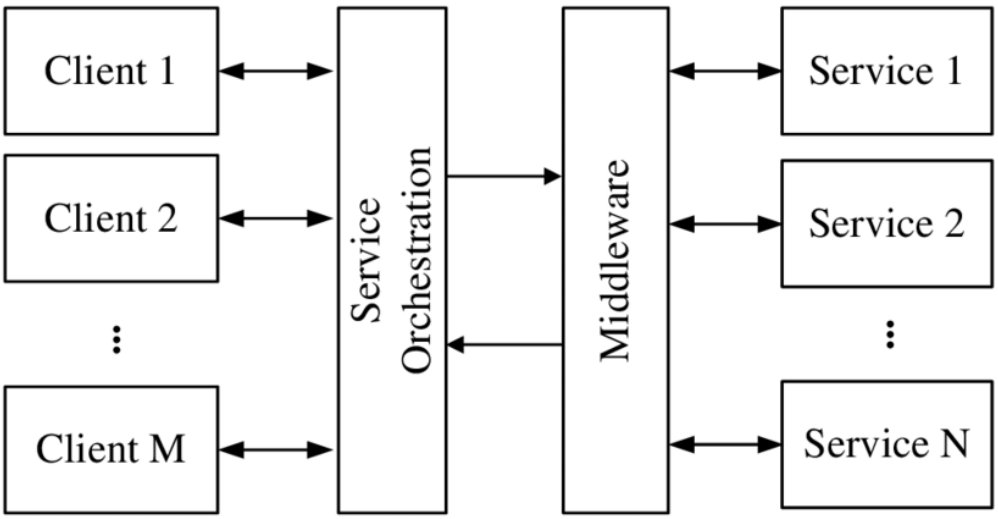
ODSOA menempatkan Orkestrasi (*Orchestration*) sebagai elemen kunci untuk mengelola interaksi antara layanan. Orkestrasi dapat didefinisikan sebagai proses otomatis yang mengkoordinasikan dan mengatur eksekusi layanan secara teratur untuk mencapai tujuan bisnis tertentu.

Dalam ODSOA, layanan disediakan sebagai fungsi-fungsi modular yang dapat digunakan oleh aplikasi dan sistem lain untuk memperoleh fungsionalitas tambahan. Layanan ini biasanya disediakan secara independen oleh unit bisnis atau departemen yang berbeda dan dapat diakses melalui jaringan.

ODSOA memiliki beberapa keuntungan, antara lain: skalabilitas, fleksibilitas, dan interoperabilitas. Skalabilitas memungkinkan sistem untuk diukur dan meningkatkan kapasitasnya dengan mudah. Fleksibilitas memungkinkan pengguna untuk menyesuaikan layanan sesuai kebutuhan mereka tanpa harus mengubah keseluruhan arsitektur. Interoperabilitas memungkinkan sistem untuk berinteraksi dengan sistem lain yang menggunakan standar yang sama.

Secara keseluruhan, ODSOA dapat membantu perusahaan dalam mempercepat pengembangan dan integrasi aplikasi serta meningkatkan efisiensi dan efektivitas bisnis secara keseluruhan.

11.2 *Orchestration-driven Service-oriented Architecture Schema*



Gambar 11.1: Arsitektur ODSOA.

Orchestration-driven Service-oriented Architecture (ODSOA) Schema adalah suatu model yang menggambarkan arsitektur ODSOA secara visual, yang mencakup komponen-komponen utama

dan interaksi antara mereka. Beberapa komponen utama dalam schema ODSOA antara lain:

- Layanan (*Services*): Komponen inti dari ODSOA adalah layanan, yang merupakan unit fungsional yang terdistribusi secara terpisah namun saling terkait secara fungsional. Layanan ini dapat digunakan oleh aplikasi dan sistem lain untuk memperoleh fungsionalitas tambahan.

- Orkestrasi (*Orchestration*): Orkestrasi merupakan proses otomatis yang mengkoordinasikan dan mengatur eksekusi layanan secara teratur untuk mencapai tujuan bisnis tertentu. Orkestrasi dapat mengatur urutan dan kondisi yang harus dipenuhi oleh layanan.

- Bus Layanan (*Service Bus*): Bus Layanan adalah infrastruktur yang memfasilitasi komunikasi antara layanan dalam arsitektur ODSOA. Bus Layanan dapat mengatur dan mengarahkan permintaan dan respons antara layanan.

- Repositori Layanan (*Service Repository*): Repositori Layanan adalah tempat untuk menyimpan informasi terkait dengan layanan yang tersedia, seperti deskripsi, spesifikasi teknis, dan interdependensi antara layanan. Repositori Layanan memungkinkan pengguna untuk mencari dan menemukan layanan yang dibutuhkan.

- Klien (*Client*): Klien adalah aplikasi atau sistem yang menggunakan layanan untuk memperoleh fungsionalitas tambahan. Klien mengirim permintaan ke layanan dan menerima respons dari layanan.

- Penyedia Layanan (*Service Provider*): Penyedia Layanan adalah unit bisnis atau departemen yang menyediakan layanan untuk digunakan oleh aplikasi dan sistem lain. Penyedia

Layanan bertanggung jawab untuk mengembangkan dan menjaga layanan yang disediakan.

Dalam ODSOA Schema, interaksi antara komponen-komponen tersebut direpresentasikan dengan panah yang menghubungkan mereka. Misalnya, panah dari klien ke layanan menunjukkan bahwa klien menggunakan layanan tersebut, sedangkan panah dari layanan ke bus layanan menunjukkan bahwa layanan terdaftar dalam infrastruktur bus layanan. Dengan ODSOA Schema, pengguna dapat dengan mudah memahami arsitektur ODSOA secara visual dan mengidentifikasi komponen-komponen utama dan interaksi antara mereka.

11.3 Kelebihan

- Skalabilitas: ODSOA memungkinkan sistem untuk diukur dan meningkatkan kapasitasnya dengan mudah. Layanan dapat dikonfigurasi ulang atau ditambahkan ke infrastruktur dengan mudah, tanpa mempengaruhi sistem keseluruhan. Hal ini memudahkan perusahaan untuk menyesuaikan sistem mereka dengan perubahan kebutuhan bisnis.

- Fleksibilitas: ODSOA memungkinkan pengguna untuk menyesuaikan layanan sesuai kebutuhan mereka tanpa harus mengubah keseluruhan arsitektur. Dengan demikian, perusahaan dapat dengan mudah memodifikasi fungsionalitas sistem dan mengintegrasikan solusi baru tanpa mempengaruhi sistem keseluruhan.

- Interoperabilitas: ODSOA memungkinkan sistem untuk berinteraksi dengan sistem lain yang menggunakan standar yang sama. Hal ini memungkinkan perusahaan untuk berintegrasi dengan sistem lain dengan mudah dan memperluas

fungsionalitas sistem mereka.

- Reusabilitas: Layanan dalam ODSOA adalah modular dan dapat digunakan kembali oleh aplikasi dan sistem lain. Hal ini memungkinkan perusahaan untuk mengembangkan sistem dengan cepat dan efisien.

- Pemisahan Tugas: ODSOA memisahkan tugas-tugas sistem menjadi layanan yang terpisah secara fisik namun saling terkait secara fungsional. Hal ini memudahkan manajemen sistem dan memungkinkan perusahaan untuk mengoptimalkan penggunaan sumber daya.

11.4 Kekurangan

- Kompleksitas: Arsitektur ODSOA dapat menjadi sangat kompleks, terutama ketika menangani banyak layanan yang berbeda dan memerlukan integrasi yang kompleks. Oleh karena itu, perusahaan memerlukan tingkat keahlian teknis yang tinggi untuk mengimplementasikan dan mengelola arsitektur ini dengan efektif.

- Keamanan: Arsitektur ODSOA dapat menimbulkan masalah keamanan karena penggunaannya yang melibatkan layanan dari banyak sistem dan vendor. Oleh karena itu, perusahaan harus memperhatikan masalah keamanan yang terkait dengan integrasi dan melakukan tindakan yang tepat untuk mengurangi risiko keamanan.

- Pengelolaan versi: Dalam ODSOA, perubahan pada satu layanan dapat mempengaruhi layanan lainnya. Oleh karena itu, pengelolaan versi menjadi penting untuk memastikan bahwa perubahan yang dibuat pada layanan tidak mengganggu kinerja sistem secara keseluruhan.

- Biaya: Implementasi arsitektur ODSOA memerlukan biaya yang tinggi karena melibatkan pengembangan, integrasi, dan manajemen layanan yang kompleks. Oleh karena itu, perusahaan harus mempertimbangkan biaya ini sebelum mengimplementasikan arsitektur ini.
- Ketergantungan terhadap vendor: Terkadang perusahaan tergantung pada vendor tertentu untuk memasok layanan tertentu. Jika vendor tersebut menghentikan layanannya, maka perusahaan perlu mencari alternatif layanan dari vendor lain atau bahkan harus mengubah arsitektur sistem secara keseluruhan.

11.5 Penerapan dalam Aplikasi

Berikut adalah contoh penerapannya dalam sebuah aplikasi

Bab 12

Microservices

Alfred Gerald Thendiwijaya, Lucky Rusandana, Inzaghi Posuma Al Kahfi

12.1 Definisi *Microservices*

Microservices adalah sebuah arsitektur perangkat lunak yang membagi sebuah aplikasi besar menjadi beberapa komponen kecil yang independen dan dapat berkomunikasi dengan satu sama lain melalui antarmuka yang didefinisikan secara jelas. Setiap komponen atau layanan (service) dalam arsitektur microservices memiliki tugas dan tanggung jawab tertentu yang dapat dijalankan secara mandiri dan dapat diubah tanpa mempengaruhi layanan lain dalam aplikasi. Dalam arsitektur microservices, komunikasi antara layanan biasanya dilakukan melalui protokol HTTP atau pesan. Kelebihan arsitektur microservices antara lain skalabilitas, fleksibilitas, dan dapat dikembangkan oleh beberapa tim yang bekerja secara terpisah.

12.2 Karakteristik *Microservices*

- Berorientasi pada layanan: Microservices dirancang sebagai layanan-layanan yang mandiri dan longgar terkait satu

sama lain, masing-masing dengan fungsionalitas dan kemampuan yang unik.

- Skalabilitas: Microservices dirancang agar mudah ditingkatkan kapasitasnya, sehingga layanan-layanan tambahan dapat ditambahkan jika ada peningkatan permintaan.

- Terdesentralisasi: Setiap microservice dapat dikembangkan dan dideploy secara independen, yang memungkinkan fleksibilitas yang lebih besar dan siklus pengembangan yang lebih cepat.

- Ketahanan: Microservices dirancang agar toleran terhadap kegagalan, dengan setiap layanan dapat beroperasi secara mandiri bahkan jika layanan lain sedang down atau mengalami masalah.

- Ringan: Setiap microservice kecil dan berfokus pada fungsi yang spesifik, yang memungkinkan pengujian, deployment, dan pemeliharaan yang lebih mudah.

- Komunikasi berbasis API: Microservices berkomunikasi satu sama lain melalui API yang ringan, menggunakan protokol seperti HTTP atau REST.

- Integrasi dan deployment berkelanjutan: Microservices sering dideploy melalui pipeline integrasi dan deployment (CI/CD) otomatis, yang memastikan bahwa perubahan dapat digulirkan ke produksi dengan cepat dan mudah.

12.3 Kelebihan *Microservices*

Berikut adalah beberapa kelebihan dari menggunakan arsitektur microservices dalam pengembangan perangkat lunak:

- Scalability: Arsitektur microservices memungkinkan skalabilitas yang lebih baik dibandingkan dengan monolithic architecture. Dalam arsitektur microservices, aplikasi terdiri dari banyak layanan yang dapat diubah ukurannya secara independen, sehingga memungkinkan untuk meningkatkan kapasitas dan throughput pada layanan tertentu tanpa harus memperbesar seluruh aplikasi.

- Fleksibilitas: Dalam arsitektur microservices, setiap layanan dapat dikembangkan secara terpisah tanpa mempengaruhi layanan lainnya. Hal ini memudahkan pengembang dalam memperbaiki, menambahkan, atau mengubah fitur pada layanan tersebut tanpa harus memperhatikan bagaimana layanan lainnya berfungsi.

- Toleransi Kesalahan: Jika terjadi kesalahan pada satu layanan, maka layanan lainnya masih dapat berjalan normal dan tidak terganggu. Hal ini memastikan bahwa aplikasi tetap berjalan dengan baik meskipun terdapat masalah pada salah satu layanan.

- Skalabilitas tim: Dalam arsitektur microservices, tim pengembang dapat fokus pada layanan tertentu dan membuat perubahan dengan cepat tanpa harus memikirkan bagaimana perubahan tersebut akan memengaruhi layanan lain dalam aplikasi. Hal ini memungkinkan untuk lebih mudah menambahkan anggota tim atau memisahkan tim kecil yang fokus pada layanan tertentu.

- Teknologi yang beragam: Dalam arsitektur microservices, setiap layanan dapat menggunakan teknologi yang berbeda. Ini memungkinkan untuk menggunakan teknologi yang paling sesuai dengan kebutuhan layanan tersebut tanpa harus mempertimbangkan teknologi yang digunakan oleh layanan lain dalam aplikasi.

- Skalabilitas bisnis: Dalam arsitektur microservices, setiap layanan dapat berjalan secara independen, sehingga memungkinkan untuk lebih mudah menambahkan fitur baru atau menghilangkan fitur yang sudah tidak diperlukan lagi. Hal ini memungkinkan bisnis untuk lebih fleksibel dalam menyesuaikan diri dengan perubahan kebutuhan pengguna dan pasar.

12.4 Kekurangan *Microservices*

- Kompleksitas: Penggunaan arsitektur microservices dapat meningkatkan kompleksitas sistem secara keseluruhan. Hal ini disebabkan karena terdapat banyak layanan yang berinteraksi satu sama lainnya, sehingga perlu perencanaan dan koordinasi yang baik dalam pengembangan.

- koordinasi lebih rumit:Akibat dari sistem yang menjadi kompleks, koordinasi antar layanan mungkin agak lebih rumit. Sebab, setiap layanan berjalan sendiri-sendiri.

- Perlu banyak automation:microservices juga membutuhkan sistem automation yang cukup tinggi untuk bisa melakukan deployment.

- Biaya: Penggunaan arsitektur microservices dapat memerlukan biaya yang lebih tinggi karena infrastruktur yang dibutuhkan lebih kompleks dan terdapat banyak layanan yang harus dikelola.

12.5 Penerapan Microservices pada aplikasi

Penerapan Microservices dalam aplikasi memungkinkan pembagian tugas dan tanggung jawab menjadi lebih terfokus, sehingga

dapat memudahkan pengembangan dan pengelolaan aplikasi secara terpisah. Setiap layanan dalam arsitektur Microservices dapat dikembangkan secara independen oleh tim yang berbeda, sehingga proses pengembangan dapat lebih cepat dan efisien. Selain itu, Microservices juga memungkinkan penggunaan teknologi yang berbeda-beda untuk setiap layanan, yang dapat meningkatkan fleksibilitas dan skalabilitas aplikasi.

12.6 Contoh penerapan

Contoh penerapan Microservices dapat ditemukan pada aplikasi e-commerce seperti Shopee dan Gojek, yang menggunakan banyak layanan terpisah untuk setiap fitur aplikasi seperti pembayaran, pengiriman, dan pemesanan. Dengan menggunakan Microservices, aplikasi dapat diintegrasikan dengan mudah dan dapat berjalan secara independen, sehingga memudahkan dalam pemeliharaan dan pengembangan aplikasi secara keseluruhan.

Bab 13

Arsitektur Continer (Container Architecture)

Richwen Canady, Desfantio Wuidjaja, Vincenzo Matalino

13.1 Latar Belakang

Konsep container berasal dari teknologi chroot pada sistem operasi UNIX. Teknologi ini memungkinkan pengguna untuk membuat lingkungan kerja yang terisolasi pada sistem operasi UNIX. Di lingkungan kerja ini, pengguna dapat menjalankan aplikasi secara mandiri tanpa terpengaruh oleh aplikasi lain yang berjalan di sistem yang sama. Namun, teknologi chroot memiliki beberapa keterbatasan, seperti pengguna harus mengkonfigurasi secara manual, tidak mendukung manajemen sumber daya.

Pada tahun 2008, LXC (Linux Containers) mengembangkan teknologi container sebagai solusi untuk mengatasi keterbatasan teknologi chroot pada sistem operasi UNIX. Teknologi container memungkinkan pengguna untuk menjalankan aplikasi secara otomatis dan efisien dalam lingkungan terisolasi yang mudah dikelola. Teknologi kontainer berjalan di sistem operasi Linux, menggunakan kernel yang sama untuk menjalankan aplikasi di dalam

container.

Pada 2013, Docker dirilis sebagai implementasi teknologi container yang lebih ramah pengguna dan mudah digunakan. Docker menyediakan gambar yang berisi semua elemen yang diperlukan untuk menjalankan aplikasi dalam container, termasuk aplikasi, sistem operasi, dan dependensi. Gambar Docker mudah dibuat, dikelola, dan dibagikan, dan dapat digunakan untuk penerapan cepat di lingkungan produksi.

Container menjadi lebih populer dan banyak digunakan untuk pengembangan dan pengelolaan aplikasi di lingkungan cloud. Container memungkinkan pengguna mengoptimalkan penggunaan sumber daya, meningkatkan portabilitas, dan mengelola aplikasi dengan mudah. Container juga mendukung orkestrasi, seperti Kubernetes, untuk mengelola aplikasi di lingkungan yang lebih kompleks. Saat ini, container adalah teknologi penting dalam pengembangan dan manajemen aplikasi.

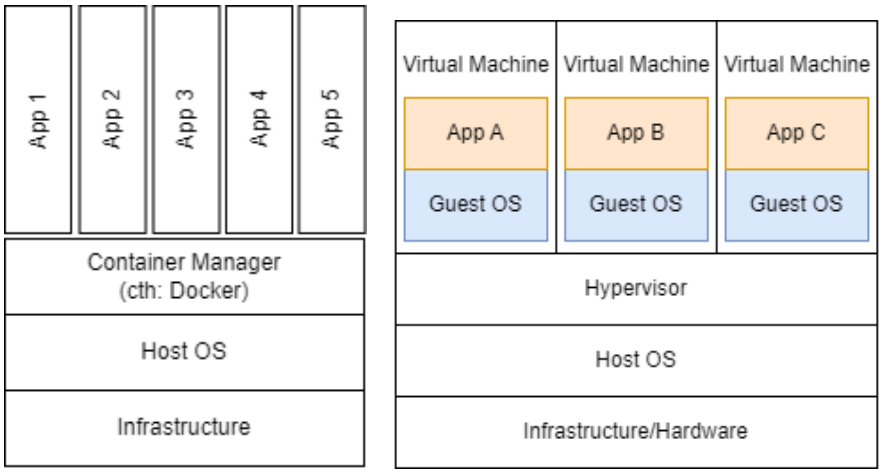
13.1.1 Virtualization vs Container Architecture

Container architecture dan *virtualization* adalah dua teknologi yang sering digunakan dalam pengembangan dan pengelolaan aplikasi, namun ada beberapa perbedaan antara keduanya yaitu:

- Isolasi= Arsitektur container menggunakan teknologi yang lebih ringan untuk menjalankan aplikasi di lingkungan yang terisolasi. Virtualisasi, di sisi lain, menggunakan teknologi hypervisor untuk mengisolasi lingkungan virtual dari sistem host. Oleh karena itu, arsitektur container lebih efisien daripada virtualisasi dalam hal penggunaan sumber daya.
- Sistem operasi= Arsitektur container menggunakan kernel yang sama dengan sistem operasi host untuk menjalankan

aplikasi dalam container. Virtualisasi, di sisi lain, memungkinkan pengguna untuk menjalankan sistem operasi yang berbeda dalam lingkungan virtual.

- Portabilitas= Arsitektur container mendukung portabilitas. Pengguna dapat mengembangkan aplikasi di lingkungan pengembangan dan dengan mudah menjalankannya di lingkungan produksi. Pada saat yang sama, virtualisasi memerlukan konfigurasi yang lebih kompleks untuk menjalankan lingkungan virtual di lingkungan produksi yang berbeda.
- Overhead= Overhead arsitektur container lebih rendah daripada virtualisasi karena tidak memerlukan overhead hypervisor dan kernel. Oleh karena itu, arsitektur container lebih efisien dalam hal penggunaan sumber daya.
- Orkestrasi= Arsitektur container memungkinkan pengguna menggunakan orkestrasi (seperti Kubernetes) untuk mengelola aplikasi di lingkungan yang lebih kompleks. Virtualisasi tidak memiliki dukungan orkestrasi yang sama.



Gambar 13.1: Arsitektur Container vs Virtualization.

13.2 Definisi

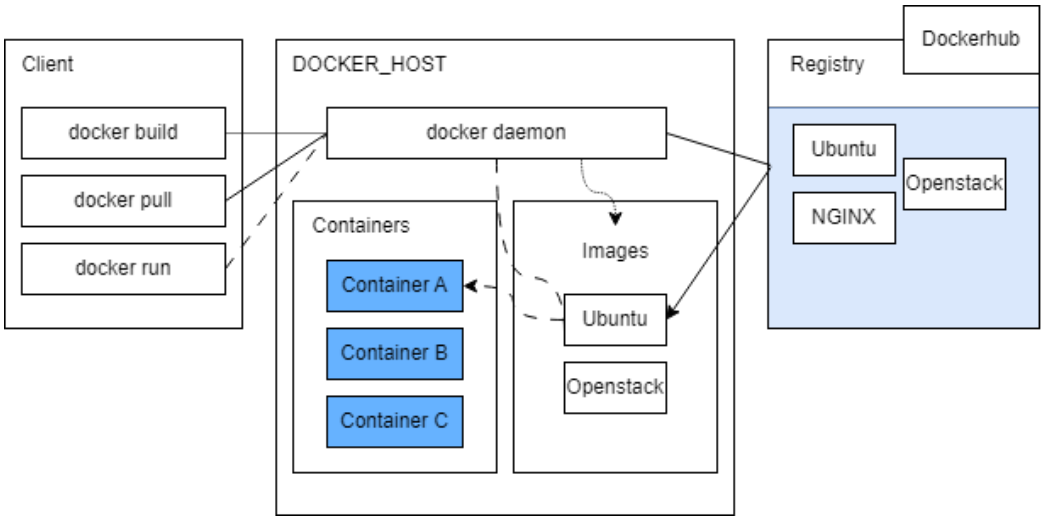
Container Architecture merupakan sebuah konsep arsitektur yang dirancang untuk menjalankan aplikasi dalam container. Container Architecture memiliki beberapa tugas yaitu Isolasi, Portabilitas, Efisiensi, Deployment.

Container adalah metode menjalankan aplikasi yang memungkinkannya berjalan secara konsisten di berbagai lingkungan komputasi.
Secara sederhana, container dapat dianggap sebagai paket yang berisi semua elemen yang diperlukan untuk menjalankan aplikasi tertentu, seperti OS, library, config, dependencies, dan file penting lainnya yang dibutuhkan untuk menjalankan aplikasi tersebut.

Docker adalah platform open source untuk mengembangkan, menguji, dan mengimplementasikan aplikasi dalam container. Dalam konteks Docker, container adalah lingkungan terisolasi yang dapat berjalan di host yang sama tanpa pengaruh aplikasi atau sistem operasi lain yang berjalan di host yang sama. Container dapat dianggap sebagai paket yang berisi semua elemen yang diperlukan untuk menjalankan aplikasi tertentu, termasuk perangkat lunak, pustaka, konfigurasi, dan dependensi lainnya.

Alternatifnya Kubernetes, adalah platform open source untuk mengelola aplikasi dalam container yang dibuat oleh Google. Kubernetes memungkinkan pengguna untuk menjalankan, mengelola, dan mengotomatiskan penerapan aplikasi dalam container secara efisien.

Docker image. Image disini bukan lah Image yang kita bayangkan (.jpg, .png, etc). Image pada Docker adalah sebuah template read-only atau cuplikan/snapshot berisikan instruksi untuk membuat container yang nantinya akan dipakai. Docker Image membuat Container untuk dijalankan di Docker Platform. Analoginya, Docker image itu seperti blueprint aparte-



Gambar 13.2: Diagram Docker.

men.

Docker Build. Cara membuat Docker image adalah dengan membuat file Dockerfile (yaitu instruksi pembuatan imagenya) dan menggunakan "docker build" pada dockerfile tersebut.

Docker Pull adalah perintah untuk mendownload/pull image docker dari registry(cth Dockerhub)

Docker Registry adalah sebuah repository berbagai docker images yang dibagikan oleh para developer.

Docker Run adalah perintah untuk menjalankan docker images dan membentuk Docker Container berdasarkan image yang dipilih.

Docker Container adalah instansi Container hasil dijalankannya image yang bisa distart, stop, restart, ataupun dihapus. Analoginya, inilah ruangan Kos apartemen hasil blueprint.

Docker Daemon adalah mesin/engine yang berjalan di mesin host dan memanage semua proses pada docker. Semua perintah perintah diatas seperti docker run itu dikirim ke docker daemon dan dijalankan.

13.3 Kelebihan dan Kekurangan

Berikut adalah kelebihan dan kekurangan docker:

13.3.1 Kelebihan

Keuntungan dari menggunakan docker adalah:

- Docker mempunyai konfigurasi yang *sederhana* yang dapat disesuaikan dengan kebutuhan aplikasi yang sedang dikembangkan. Dengan menetapkan beberapa baris kode yang mendukung, docker mampu membuat lingkungannya sendiri yang terpisah dari lingkungan server utama.

- Docker memiliki tingkat keamanan yang baik. Ia akan memastikan aplikasi yang sedang berjalan tidak dapat memengaruhi container *(isolation)*. Selain itu, ia juga memiliki fitur keamanan *pengaturan OS host mount* dengan akses *read-only* sehingga konfigurasi yang tersedia tidak akan berubah sama sekali (kecuali pengguna memiliki akses penuh).

- Docker dapat dijalankan pada beberapa platform cloud. Karena itu, pengguna dapat melakukan porting aplikasi dengan lebih mudah dan fleksibel. Selain itu, fitur-fitur docker juga dapat dijalankan pada berbagai sistem operasi, seperti Windows, Mac, dan Linux.

- Docker mempunyai ukuran yang cukup ringan, dan lebih hemat sumber daya. Pengguna tidak membutuhkan memory storage atau overhead yang terlalu besar untuk menggunakannya.

- Docker memiliki fitur debugging. Waktu yang dibutuhkannya juga tergolong cepat, yakni hanya sekitar satu menit saja untuk melakukan proses debug pada Sandbox.

13.3.2 Kekurangan

Konsekuensi dari menggunakan docker adalah sebagai berikut:

- Walaupun dapat digunakan pada berbagai macam OS, docker mempunyai kompatibilitas *cross-platform* yang kurang fleksibel. Ketika sebuah aplikasi dirancang menggunakan Windows, pengguna memerlukan bantuan *tools* eksternal untuk menjalankannya di Linux.
- Secara garis besar, docker memiliki kekurangan fitur yang harus diakali pengguna dengan cara meng-install perangkat lunak eksternal apabila pengguna tidak ingin melakukan manajemen manual. Contohnya, docker tidak mempunyai dukungan untuk health-check, atau pemrograman ulang otomatis dari node yang tidak aktif.

13.4 Contoh Kasus Penggunaan Container Architecture

Biasanya, Container Architecture ergo Docker Container dibutuhkan dalam pembuatan dan deploy aplikasi yang terdiri dari beberapa komponen berbeda, seperti aplikasi web yang terdiri dari server web, database, dan layanan lainnya.

Dengan menggunakan Docker, kita dapat mengemas setiap komponen aplikasi ke dalam container yang terisolasi dan dapat dijalankan secara independen di berbagai lingkungan, seperti lingkungan pengembangan, pengujian, dan produksi. Container Docker memungkinkan pengembang untuk menjamin bahwa aplikasi yang mereka kembangkan dapat dijalankan dengan konsisten di seluruh lingkungan, sehingga mengurangi risiko terjadinya kesalahan dan masalah ketika aplikasi dideploy.

Tanpa container/docker, dalam pembuatan aplikasi kita biasanya harus install dan konfigurasi setiap komponen aplikasi se-

cara manual di setiap environment, seperti environment pengembangan, pengujian, dan produksi. Ini bisa bermasalah ketika aplikasi dideploy di lingkungan yang berbeda, karena berbeda konfigurasi dan pengaturannya.

Contoh kasus, misal ada sebuah aplikasi php yang sudah didevelop menggunakan php7, belum tentu aplikasi tersebut bisa dijalankan di komputer lain yang menjalankan php5. Dengan menggunakan docker, meskipun pada dasarnya komputernya menggunakan php5, namun image dan containernya sudah ada php7 jadinya tidak perlu konfigurasi ulang.

13.5 Demo Container Architecture Menggunakan Docker

Pada bagian ini Richwen akan mendemokan cara kerja docker. Codenya ada di folder code chapter 13.

Bab 14

DevOps

Hendra Lijaya, Oktavianus Hendry Wijaya

14.1 Pengertian

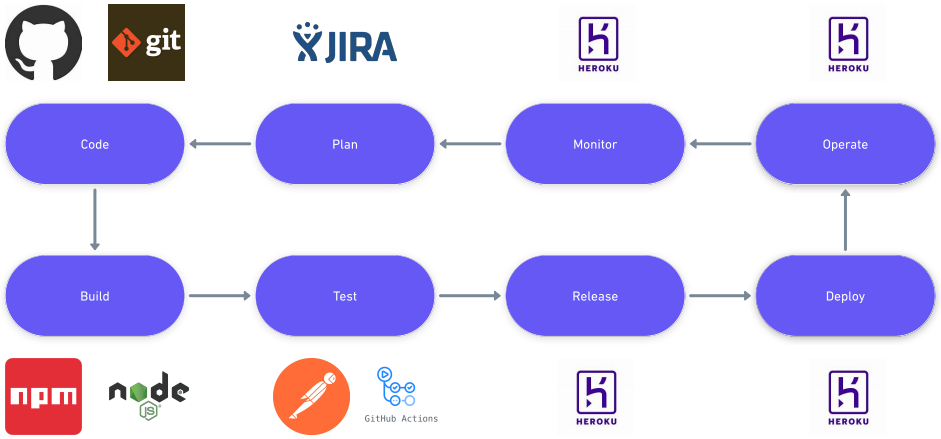
DevOps merupakan metode pengembangan software dengan mengkolaborasikan *software developer* dengan *IT operation*.

14.2 Fungsi

Tujuan akhir atau *goal* dari DevOps adalah untuk menciptakan lingkungan kolaborasi yang berkelanjutan untuk membawa software menjadi lebih berkualitas, lebih cepat, dan dapat diandalkan.

14.3 Arsitektur

- *Plan*
  Tahap paling awal dalam SDLC (*Software Development Life Cycle*). Mulai dari tahap pengumpulan data, membuat *roadmap*, menetapkan tujuan, *timelines* serta mengidentifikasi *resources* yang diperlukan untuk menyelesaikan sebuah proyek.



Gambar 14.1: Arsitektur DevOps.

- *Code*
  Pada tahap ini, developer mulai menulis kode untuk mengembangkan *software* berdasarkan requirement yang telah dikumpulkan pada tahap *Plan*.

- *Build*
  Pada tahap ini, kode di *compile*, dan di *package* kedalam format yang bisa di *deliver*. Tujuan tahap ini membuat kode yang telah di *compile* agar dapat melakukan tahap *Testing* dan *Release*.

- *Test*
  Pada tahap ini, perangkat lunak diuji untuk memastikan bahwa perangkat lunak sudah memenuhi requirement dan fungsinya sudah berjalan tanpa ada *bug*.

- *Release*
  Pada tahap ini, perangkat lunak sebelum di *deploy* ke *staging* atau *production environment* dapat dilakukan data migrasi, konfigurasi dan lainnya. Tujuannya adalah untuk memastikan bahwa perangkat lunak tersedia dan dapat digunakan oleh audiens yang dituju.

- *Deploy*
  Pada tahap ini, perangkat lunak di *deploy* ke *production* ataupun *staging* bisa menggunakan tools atau *automation script*. Proses ini mencakup juga instalasi library, dan konfigurasi server dan lainnya.

- *Operate*
  Pada tahap ini, perangkat lunak sudah beroperasi di *production environment* dan dapat dikelola dan dipantau untuk memastikan kinerjanya seperti yang diharapkan.

- *Monitor*
  Pada tahap ini, pengumpulan dan analisis data dari log sistem. Informasi ini dapat digunakan untuk mengidentifikasi area untuk perbaikan, mengoptimalkan kinerja, dan menginformasikan upaya pengembangan perangkat lunak kedepannya.

14.4 Kelebihan Kekurangan

Berikut adalah kelebihan dan kekurangan arsitektur DevOps:

14.4.1 Kelebihan

Keuntungan dari menerapkan arsitektur DevOps adalah:

- DevOps menjadi pilihan yang bagus untuk *development* dan *deployment* aplikasi yang cepat

- Merespon lebih cepat ke perubahan market untuk meningkatkan *business growth* (pertumbuhan bisnis)
- Data dapat disentralisasikan sehingga meningkatkan konsistensi data dan mengurangi duplikasi data.
- DevOps meningkatkan profit bisnis dengan mengurangi waktu *delivery software* dan biaya.
- DevOps menghilangkan proses deskriptif sehingga memberikan kejelasan mengenai *development* dan *delivery product*.
- Meningkatkan *experience* dan kepuasan *customer*.
- DevOps menyederhanakan kolaborasi dan menggunakan semua *tools* di cloud untuk diakses pengguna.
- Meningkatkan keterlibatan dan produktivitas tim.

14.4.2 Kekurangan

Kekurangan dari penerapan arsitektur DevOps adalah:

- DevOps *professional* atau *expert* masih belum umum ditemukan.
- *Developing* dengan DevOps mahal.
- Penerapan DevOps baru ke dalam industri sulit untuk dikelola dalam waktu singkat.
- Kurangnya pengetahuan mengenai DevOps dapat menyebabkan masalah pada *Continuous Integration* dari *project automation*.

14.5 Perbedaan DevOps dan nonDevOps

Perbedaan besar dari *development* dengan DevOps dan nonDevOps yaitu:

1. Kolaborasi
   Pada *Software development* nonDevOps, developer dan tim operation bekerja secara terpisah. Sedangkan pada DevOps, kedua hal tersebut bekerja secara kolaboratif dalam 2 tim yang berbeda dengan berbagi pengentahuan dan skill sehingga memastikan proses software development dapat disederhanakan.

2. *Continuous Integration and Delivery (CI/CD)*
   DevOps menerapkan CI/CD yang dimana melibatkan sistem *automation* pada proses *software development* mulai dari *building* dan *testing* hingga ke *deployment* dan *maintenance*. Dengan menerapkan ini, perubahan dapat di tes dan diintegrasikan ke software secara cepat dan efisien mungkin.

3. *Automation*
   DevOps bergantung secara penuh pada automation untuk meningkatkan efisiensi dan mengurangi error. Tools seperti *management configuration*, *continuous integration*, dan *continuous delivery* memungkinkan tim untuk mengautomatis proses yang awalnya manual dan memastikan konsistensi.

4. *Monitoring*
   Tim yang menerapkan DevOps menggunakan *tools* untuk *monitoring* dan analitik untuk mengumpulkan data mengenai performa pada software saat *production*. Hal ini membantu tim dalam mengidentifikasi dan menyelesaikan isu dengan cepat sehingga mengurangi downtime dan meningkatkan *user experience* secara keseluruhan.

5. Agile Development
   DevOps berdasarkan pada prinsip *agile development* yang dimana fleksibilitas, adaptabilitas, dan kolaborasi sangat ditekankan. Tim DevOps memprioritaskan dalam *delivery* dalam perubahan kecil dan perubahan *incremental* dengan cepat dibandingkan perilisan monolitik yang bersifat besar.

Kesimpulannya adalah DevOps bersifat lebih kolaboratif, *automated*, dan *agile* pada proses *software development* yang menekankan *continuous integration and delivery, automation,* dan *monitoring.*

14.6 Tools

Tools yang digunakan dalam pembuatan DevOps:

1. Git - GitHub Action
   GitHub Action adalah fitur dari *platform* GitHub yang memungkinkan developer untuk mengautomasi *workflows* dan *build, test*, dan *deploy* kode langsung dari *platform* GitHub. GitHub Action menyediakan *library* dari *pre-built actions* yang dapat digunakan untuk membangun *workflows* dan juga kemampuan untuk membuat action kustom menggunakan JavaScripts atau Docker containers. *Workflows* dapat dipicu/di*trigger* oleh *events* seperti push kode, request pull, atau pembuatan perilisan baru.
   Keuntungan menggunakan GitHub:

   - Terintegrasi dengan GitHub
   - Workflows yang dapat dikustomisasi
   - Reusability
   - Kolaborasi
   - Skalabilitas

- Gratis

Kesimpulan, GitHub Action merupakan *tools* yang sangat berguna untuk *automating software development workflows*, menyediakan developer fleksibilitasm, kustomisasi, dan platform yang terintegrasi untuk *building, testing,* dan *deploy* kode

2. Heroku
Heroku merupakan *platform cloud* yang memungkinkan developer untuk *build, deploy*, dan mengelola aplikasi secara cepat dan mudah. Heroku mendukung beberapa Bahasa pemrograman seperti Java, Ruby, Node.js, Python, PHP, dan Go. Heroku menyediakan platform yang dikelola secara penuh sehingga developer tidak perlu mengkhawatirkan mengenai mengelola infrastruktur, sistem operasi, dan server.
Heroku didasarkan pada arsitektur yang berbasis *container* dan menggunakan Dyno untuk menjalankan aplikasi. Dyno merupakan *container* linux yang ringan dan terisolasi yang berjalan diatas platform Heroku. Dyno di*design* untuk menjalankan satu proses atau layanan yang membantu meningkatkan performa, skalabilitas, dan ketahanan.
Fitur-fitur Heroku:

   - *Command Line Interface* (CLI)
   - *Web Based Dashboard*
   - Beragam *add-ons* dan *extensions*
   - *Support continuous integration and continuous delivery (CI/CD) workflows.*

   Adapun kekurangan dari Heroku yaitu:

   - Kustomisasi yang terbatas.
   - Bergantung pada *add-on third party.*

- Memerlukan biaya dan kartu kredit.

3. Postman
   Postman merupakan tools software yang sering digunakan oleh developer untuk *test*, dokumentasi, dan berbagi API. API atau *Application Programming Interfaces* memungkinkan software aplikasi yang berbeda untuk berkomunikasi melalui pertukaran data antara satu dengan yang lain.
   Dengan menggunakan Postman, developer dapat dengan mudah membuat dan mengeksekusi *HTTP requests*, yang memungkinkan mereka untuk mencoba API dan memastikan API berjalan dengan benar. Postman juga menyediakan berbagai fitur yang memudahkan pendokumentasian API, termasuk kemampuan untuk membuat dokumentasi API dan membuat *code snippets* dalam berbagai variasi bahasa pemrograman.
   Fitur utama pada Postman:

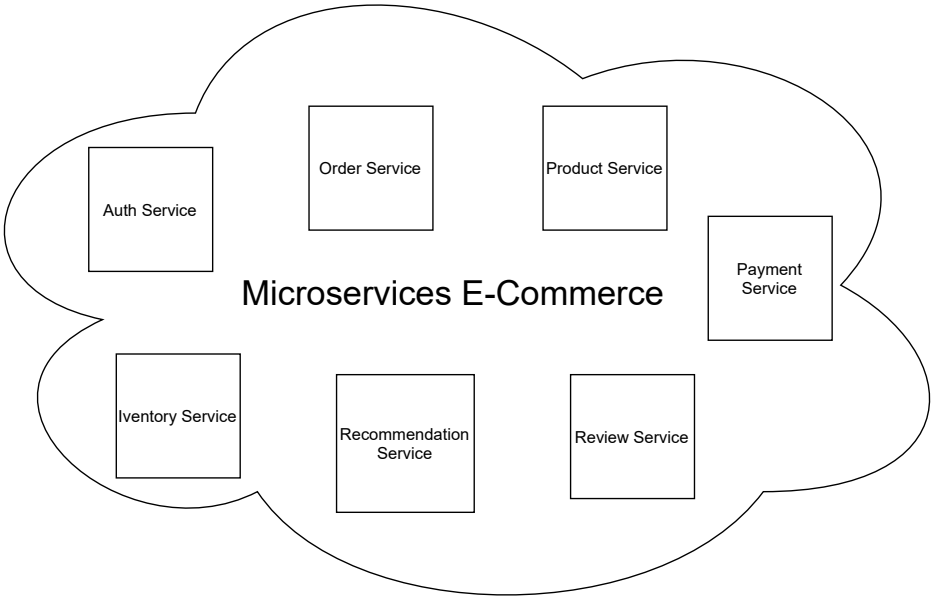
   - *Collections*: Postman memungkinkan developers untuk mengorganisasikan *request* ke sebuah *collections*, yang dimana memudakan dalam *grouping request* berdasarkan fungsionalitas.
   - *Environments*: Postman mendukung penggunaan *environments* yang memungkinkan developer untuk pendefinisian variabel dan *values* yang berbeda untuk *testing environments* yang berbeda (seperti *development, testing,* atau *production*).
   - *Test automation*: Postman memungkinkan developer untuk mengautomatisasi *testing* dengan membuat *scripts* yang dapat dijalankan sebagai bagian dari *test*. Hal ini memudahkan dalam memastikan API berjalan secara benar dan memudahkan mencari isu diawal dalam proses *development*.

- *Collaboration*: Postman memudahkan dalam berkolaborasi dengan developer lain dengan memungkinkan berbagai *collection, environment,* dan *documentation* dengan yang lain.

- *Integrations*: Postman terintegrasi dengan bermacam-macam tools dan layanan lain, seperti GitHub, Jira, Slack, yang memudahkan dalam memasukkan Postman kedalam *workflows* yang sudah ada.

14.7 Contoh Kasus

Contoh Kasus Aplikasi *E-Commerce* Biasanya pada aplikasi *e-commerce* yang menggunakan arsitektur *microservice.* Beberapa *service* pada *microservice* yaitu *Authentication Service, product catalog service, order management service, payment service, and shipping service,* yang masing-masing bertanggung jawab untuk fungsi tertentu. Dengan menggunakan DevOps dapat memberikan beberapa kemudahan pada developer dengan beberapa fitur.
Fitur utama DevOps:
CI/CD Pipeline digunakan untuk melakukan *compile code*, *unit testing*, *integration testing*, *packaging*, *deployment* dan *monitoring.* Setiap kali developer melakukan *push* pada *repository* git, *CI/CD Pipeline* otomatis melakukan *compile code*, menjalankan *unit tests*, *deploy* perubahan pada *staging environment* untuk *integration testing.* Jika *integration test* dilewati, maka akan di *deploy* di *production environment.*
Menggunakan DevOps dapat membuat developer fokus pada fitur aplikasi yang ingin dibangun sehingga tidak terlalu lama memikirkan setup infrastruktur untuk *testing* dan lainnya.



Gambar 14.2: Studi Kasus *E-Commerce Microservice*.

14.8 Code

Code dapat dilihat di link berikut
`https://github.com/alfa-yohannis/software-architecture/tree/main/code/chapter14`

14.9 Video Tutorial

Video penjelasan mengenai proses Heroku:
`https://youtu.be/My2MOkgRPwo`

Bibliography