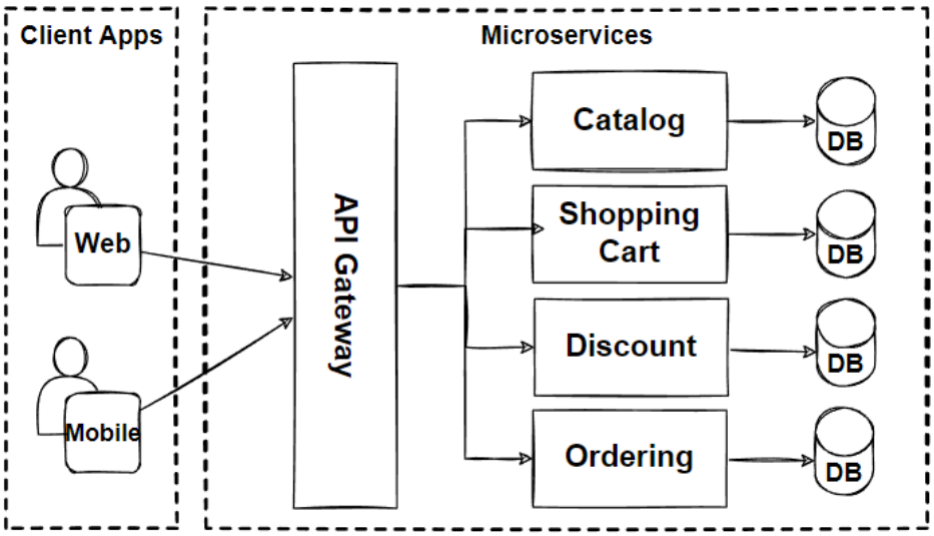
BAB 2

Landasan Teori

2.1 *Microservices*

Microservices adalah desain arsitektur aplikasi yang memecah aplikasi menjadi *service – service* kecil yang terpisah sesuai dengan fungsinya. *Service* akan dibagi menjadi lebih rinci lagi dari segi fungsionalitasnya agar setiap fungsi bekerja secara independen [1]. Dalam hal ini setiap *service* mungkin saja dapat mempunyai teknologi yang berbeda.



Gambar 2.1 *Microservices*

2.2 Web*service*s

*Web service*s adalah aplikasi perangkat lunak yang disediakan melalui internet dan dapat digunakan oleh aplikasi lain, terlepas dari platform atau bahasa pemrog*RAM*an yang digunakan [5]. Mereka menyediakan antarmuka yang terstandarisasi untuk berkomunikasi dan berinteraksi dengan aplikasi lain menggunakan protokol web seperti *HTTP* dan *XML*.

2.3 *JSON*

JSON (JavaScript object notation) merupakan format yang menyimpan informasi terstruktur dan biasanya digunakan untuk mentransfer data antara *server* dengan klien [6]. Kelebihan *JSON* dibandingkan format pertukaran informasi yang lainnya adalah dari segi struktur datanya yang sederhana dan mudah dipahami. Selain itu *JSON* tidak hanya dapat digunakan pada bahasa pemrograman JavaScript tetapi juga mendukung pada bahasa pemrograman yang lain seperti PHP, Python, Ruby, C++, dan lain lain

2.4 *REST*

REST (Representational State Transfer) merupakan protokol standar dalam suatu arsitektur *web service* [7]. Dalam hal ini *REST* menggunakan protocol *HTTP* (Hypertext Transfer Protocol) dalam melakukan proses pertukaran data antara *client* dan *server*. Dengan demikian interaksi yang terjadi antara *client* dan *server* menggunakan interface yang dapat diakses oleh protokol *HTTP*, protokol ini sering disebut juga sebagai *API* (Application Prog*RAM*ming Interface). Saat mengakses suatu *resource*, *REST* menggunakan konsep URI (Uniform *Resource* Identifier) yang memungkinkan satu *service* memiliki alamat yang berbeda dengan *service* lain dalam *server* yang sama.

2.5 *HTTP*

Sebuah aplikasi web berinteraksi dengan klien melalui protokol *HTTP* (Hypertext Transfer Protocol) [8]. Dengan menggunakan metode permintaan dan tanggapan (*request* dan *response*), *HTTP* memungkinkan aplikasi web untuk menghasilkan konten yang diakses oleh pengguna. Dalam pengembangan aplikasi web, penting bagi aplikasi untuk memiliki mekanisme yang jelas dalam membaca permintaan *HTTP* dari klien dan memberikan tanggapan *HTTP* kepada pengguna.

HTTP Request adalah proses di mana *server* membaca informasi yang dikirimkan oleh klien melalui aplikasi web *server*. Sementara itu, *HTTP Response* adalah respons dari *server* terhadap permintaan yang telah dikirimkan oleh klien. Dalam intinya, interaksi antara klien dan *server* melalui protokol *HTTP* membentuk dasar komunikasi untuk menyediakan konten web kepada pengguna.

2.6 *Domain Driven Design*

Domain – Driven Design (DDD) merupakan pendekatan pembangunan perangkat lunak dalam *Model*ing suatu aplikasi, DDD membantu dalam proses pengembangan perangkat lunak menjadi lebih akurat sesuai dengan proses bisnis yang akan berjalan pada aplikasi [4]. Salah satu masalah yang sering muncul dalam proses merancang aplikasi adalah komunikasi antar class atau fungsi yang seiring berkembangnya aplikasi maka akan semakin besar. Masalah ini akan menimbulkan *dead code* yaitu saat kode aplikasi yang besar tidak berani disentuh developer karena takut akan menimbulkan kesalahan. Semakin lama *dead code* akan semakin besar dan bahkan bertambah, hal ini menjadi sesuatu yang menakutkan bagi developer.

DDD berusaha menjadi solusi dari masalah tersebut dengan menggunakan *Bounded Context*. Aplikasi akan dipecah ke dalam konteks yang berbeda-beda sesuai dengan sub-*Domain*, selanjutnya setiap konteks seminimal mungkin dalam berkomunikasi dengan konteks lain. Proses komunikasi antar konteks harus dilakukan secara terbatas karena faktor inilah yang membuat aplikasi lebih mudah untuk di-*refactor*. Di dalam metode ini terdapat tahapan yang perlu dipahami sebelum menggunakannya pada suatu *software*. Adapun tahapan-tahapan tersebut adalah sebagai berikut:

2.6.1 Strategic Design

Strategic Design dalam konteks *Domain-Driven Design* (DDD) adalah pendekatan yang memfokuskan pada pe*Model*an dan desain perangkat lunak yang berorientasi pada *Domain* bisnis. Berikut komponen yang terdapat pada strategic design:

1. *Ubiquitous Language*

Bahasa yang digunakan oleh semua stakeholder, termasuk pengembang dan pengguna bisnis, untuk memastikan bahwa semua pihak memiliki pemahaman yang sama tentang *Domain*.

2. *Bounded Context*

Bounded Context merupakan batasan dari sebuah konteks dalam menerapkan suatu *Model*. Fungsi utama dari *Bounded Context* adalah untuk menjaga agar sebuah *Context* tetap konsisten dalam menggunakan *Model*nya sesuai dengan

batasan yang sudah ditetapkan. Sehingga ketika dalam pengembangannya atau perubahannya ada batasan yang jelas dan tidak boleh dilanggar pada setiap *Context*. Kelebihan dari diterapkannya *Bounded Context* menjadikan kita tahu, tim mana yang bertanggung jawab terhadap sebuah *Model* dan mana yang tidak.

3. *Context Map*

Dalam sebuah aplikasi yang sudah besar dan kompleks, terkadang terdapat beberapa *Model* sama yang digunakan oleh lebih dari satu *Bounded Context*. Sebuah tim dalam suatu *Context* juga tidak selalu mengerjakan seluruh bagian di dalam *Bounded Context*, sering kali bagian tersebut merupakan sub-bagian *Context* dari tim yang berbeda dan tidak jarang juga yang merupakan bagian dari pihak ketiga. Tim yang tidak mengetahui tentang *Model* dari *Context* yang berbeda demikianlah yang perlu diperhatikan. Hal ini akan memunculkan risiko tim untuk mengerjakan *Bounded Context* yang sudah dibatasi sebelumnya. Dengan kata lain batasan antara *Model* harus jelas sehingga tidak terjadi kebingungan pada *Context*.

Untuk mengatasi masalah ini dibutuhkanlah suatu cara untuk menggambarkan hubungan dan batasan dari setiap *Bounded Context* secara keseluruhan yang berupa *Context Map*. *Context map* berfungsi memastikan jika suatu tim dalam sebuah *Bounded Context* mengetahui keseluruhan dari sistem baik secara teknis atau prosesnya. Hal ini berguna untuk mengurangi masalah yang nantinya akan ditimbulkan seperti menggunakan *Model* yang melebihi dari batasan yang sudah ditentukan. Dalam membuat *Context* map terdapat jenis relasi antara masing-masing *Bounded Context*, berikut adalah jenis relasi yang terdapat dalam *Context map*:

1. Anticorruption Layer

Pada relasi dengan jenis *anticorruption layer* digunakan ketika suatu *Context* membutuhkan relasi dengan *Context* lain, tetapi *Context* tujuan tidak diizinkan untuk berkomunikasi secara langsung dengan *Context* awal. Hal ini dilakukan karena *Context* tujuan perlu untuk tetap menjaga integritas data yang dimiliki. Dengan adanya *anticorupption layer* akan menjadikan *Context* tetap terisolasi dari *Context* lain dan juga menjaga *Model* untuk tidak berubah.

2. Shared Kernel

Suatu *Context* dalam sebuah *web services* mungkin saja akan saling berkomunikasi dengan *Context* lain, proses komunikasi ini tidak hanya saling bertukar data saja namun juga logika *Model* dan hal yang lainnya. Untuk menjaga *Context* tetap terisolasi dari *Context* lain dapat menggunakan *shared kernel*. *Shared kernel* berfungsi untuk membagikan data yang dibutuhkan oleh *Context* lain dan sebaliknya dalam memudahkan proses integrasi.

3. Open Host Service

Mirip seperti *anticorruption layer* namun perbedaan yang terjadi terdapat pada *Context* yang membutuhkan data. Pada *anticorruption layer* proses komunikasi dilakukan melalui layer yang sudah terisolasi untuk menjaga integritas, meskipun data dari *anticorruption layer* sudah diterima akan sulit untuk memproses data ketika banyak *Context* yang membutuhkan data tersebut. *Open Host Service* digunakan untuk mentransformasi data dari *Context* yang terisolasi untuk dapat diproses oleh *Context* yang membutuhkan.

4. Separate Ways

Pada jenis *separate ways* proses komunikasi antar *Context* akan dilakukan tanpa sama sekali berelasi dengan *Context* lain. Dalam menentukan relasi suatu *Bounded Context* sering kali membutuhkan proses yang panjang baik dari segi bisnis maupun aturan pada masing-masing *Context* yang berhubungan. *Separate ways* berguna menjadi penengah dari hal tersebut sehingga meskipun suatu *Context* tidak berelasi dengan *Context* lain, proses komunikasi data dapat terjadi secara manual atau juga melalui *user interface* (UI) dari aplikasi itu sendiri.

5. Partnership

Partnership digunakan ketika dua buah tim atau lebih bekerja dengan *Context* yang berbeda akan tetap mempunyai tujuan *Context* yang sama. Hal ini berguna untuk memastikan kerja sama antar tim tersebut dapat terjalin. Kerja sama ini akan menimbulkan keuntungan baik dari segi integrasi dari masing-masing *Context* nantinya maupun mempercepat proses politik baik dari aturan dan yang lainnya dari masing-masing tim.

6. Downstream

Pada *Context* yang menggunakan relasi *downstream* menggambarkan jika *Context* tersebut yang membutuhkan data dari *Context* lain.

7. Upstream

Pada *Context* yang menggunakan relasi *downstream* menggambarkan jika *Context* tersebut yang membutuhkan data dari *Context* lain.

8. Customer-Supplier

Saat dua buah tim dengan tujuan *Context* yang berbeda saling berhubungan akan besar kemungkinannya bagi tim yang menjadi *upstream* untuk melakukan keputusan yang merugikan bagi tim yang menjadi *downstream*. Kerugian ini dapat mengakibatkan pekerjaan dari tim sebagai *downstream* menjadi tidak sesuai. Untuk mengatasi masalah tersebut dapat digunakan jenis relasi *customer-supplier*, pada sisi *Context* yang menjadi *downstream* akan berperan sebagai *customer* sedangkan *supplier* diisi oleh *Context* yang menjadi *upstream*-nya.

9. Conformist

Saat proses komunikasi *Context* terjadi terkadang *Context* yang berperan sebagai *upstream* tidak mengizinkan *Context* sebagai *downstream* untuk ikut berkolaborasi. Sehingga proses integrasi *Context* membutuhkan verifikasi terlebih dahulu untuk memastikan jika *Context* sebagai *upstream* tidak akan mengubah *API* tanpa pemberitahuan terlebih dahulu jenis relasi ini dinamakan dengan *conformist relationship*. Proses integrasi ini biasanya dilakukan ketika *Context* akan berhubungan *Context* lain dari *third-party*.

2.6.2 *Tactical Design*

Tactical Design adalah bagian dari DDD yang berfokus pada implementasi teknis dan konkret dari *Model Domain* yang sudah dirancang pada tahap Strategic Design. *Tactical Design* menyediakan pola-pola dan praktik-praktik yang membantu pengembang dalam membangun sistem yang sesuai dengan *Model Domain* dan memenuhi kebutuhan bisnis. Beberapa konsep utama dalam *Tactical Design* adalah:

1. Entitas

Entities adalah objek yang memiliki identitas unik yang konsisten selama masa hidupnya. Identitas ini membedakan satu entitas dari entitas lainnya, meskipun atribut-atribut mereka mungkin sama. Contohnya, dalam sistem e-commerce, entitas seperti "Pelanggan" atau "Pesanan" memiliki identitas unik.

2. *Value Objects*

Value Objects adalah objek yang sepenuhnya ditentukan oleh atribut-atributnya dan tidak memiliki identitas unik. Mereka biasanya digunakan untuk menggambarkan aspek atau deskripsi dari sesuatu, seperti "Alamat" atau "Warna". *Value Objects* tidak berubah dan sering kali dapat dioperasikan secara aman melalui cara-cara seperti komparasi atau penggabungan.

3. *Aggregates*

Aggregates adalah kumpulan entitas dan *Value Objects* yang dikelompokkan bersama-sama karena mereka membentuk satu unit konsistensi dan modifikasi. *Aggregate* memiliki satu root (root entity) yang bertanggung jawab untuk memastikan konsistensi dalam *aggregate* tersebut. Contoh *aggregate* dalam sistem e-commerce adalah "Pesanan" yang mungkin terdiri dari beberapa entitas "Item Pesanan" dan *Value Objects* seperti "Alamat Pengiriman".

4. Repositories

Repositories adalah abstraksi untuk mengakses dan memanipulasi koleksi entitas. Mereka menyediakan antarmuka untuk operasi CRUD (Create, Read, Update, Delete) tanpa harus mengetahui detail implementasi penyimpanan data. Dengan menggunakan repositories, pengembang dapat mengakses data *Domain* secara konsisten dan terisolasi dari logika penyimpanan.

5. *Service*s

*Service*s adalah operasi yang tidak secara alami cocok sebagai metode pada entitas atau *Value Objects*. Mereka biasanya digunakan untuk mengenkapsulasi logika bisnis yang tidak berkaitan langsung dengan atribut atau identitas objek *Domain* tertentu. Contoh *service* dalam sistem e-commerce adalah "Pembayaran*Service*" yang menangani logika pembayaran dan tidak tergantung pada identitas entitas tertentu.

6. Factories

Factories adalah objek atau metode yang bertanggung jawab untuk membuat instance dari entitas atau *Value Objects*. Mereka membantu mengelola kompleksitas pembuatan objek dengan mengenkapsulasi logika inisialisasi dan memastikan bahwa objek yang dibuat memenuhi semua invarian yang diperlukan.

2.7 UML (Unified *Model*ing Language)

Dalam pengembangan perangkat lunak, pe*Model*an *Domain* merupakan langkah krusial untuk menentukan elemen-elemen yang akan ada dalam sistem. Salah satu metode yang sering digunakan untuk pe*Model*an ini adalah *Class Diagram* [9]. *Class Diagram* adalah alat yang digunakan untuk menggambarkan struktur dan hubungan antar kelas dalam sebuah sistem. Dengan menggunakan *Class Diagram,* pengembang dapat memvisualisasikan struktur sistem secara menyeluruh, yang mencakup kelas, atribut, operasi, dan relasi di antara kelas-kelas tersebut.

1. Nama Kelas

Kelas adalah konsep yang menggambarkan sekelompok objek dengan karakteristik yang sama. Penamaan kelas harus mencerminkan kelompok objek yang dimaksud. Nama kelas biasanya bersifat umum dan menggambarkan kategori objek. Misalnya, dalam sistem otomotif, kelas "Mobil" dapat mencakup objek seperti BMW, Mercedes, dan Ferrari. Dengan demikian, penamaan kelas yang tepat memungkinkan pengelompokan objek yang relevan dan memudahkan pemahaman struktur sistem.

2. Atribut

Atribut adalah ciri atau karakteristik dari suatu kelas. Atribut memberikan informasi tambahan tentang objek yang termasuk dalam kelas tersebut. Beberapa aspek penting dalam mendefinisikan atribut adalah sebagai berikut:

a. Visibilitas: Menunjukkan sejauh mana atribut dapat diakses oleh kelas lain. Visibilitas atribut dapat dibagi menjadi:

b. Private: Atribut hanya dapat diakses di dalam kelas yang mendeklarasikannya.

c. Protected: Atribut dapat diakses oleh kelas yang berhubungan dan kelas yang mewarisinya.

d. Public: Atribut dapat diakses oleh kelas lain tanpa pembatasan.

e. Nama: Nama atribut biasanya pendek, diawali dengan huruf kecil, dan menggunakan huruf k*API*tal untuk awal kata berikutnya, mengikuti konvensi penamaan yang konsisten.

f. Tipe: Tipe atribut menentukan klasifikasi data yang digunakan, seperti int, string, atau boolean, sesuai dengan bahasa pemrog*RAM*an yang dipilih.

2.8 *Event Storming*

Event Storming adalah teknik pe*Model*an yang digunakan dalam *Domain-Driven Design* (DDD) untuk menggali dan memahami kompleksitas suatu *Domain* dengan cara yang kolaboratif dan *visual*. Teknik ini pertama kali diperkenalkan oleh Alberto Brandolini pada tahun 2013 [10]. *Event Storming* membantu tim pengembang perangkat lunak, pakar *Domain*, dan pemangku kepentingan lainnya untuk bersama-sama mengeksplorasi *Domain* bisnis melalui identifikasi dan analisis peristiwa (*event*s) yang terjadi dalam *Domain* tersebut.

1. *Domain Event*s

*Domain Event*s adalah peristiwa penting yang terjadi dalam *Domain* bisnis. Mereka menggambarkan perubahan keadaan yang bermakna. Contoh *Domain event* dalam sistem e-commerce termasuk "Pesanan Dibuat", "Pembayaran Berhasil", dan "Barang Dikirim".

2. Commands

Commands adalah tindakan yang diminta oleh pengguna atau sistem lain untuk memicu *Domain event*. Mereka mewakili niat untuk mengubah status dalam *Domain*. Misalnya, "Buat Pesanan" atau "Proses Pembayaran".

3. *Aggregates*

Aggregates adalah kumpulan entitas dan *Value Objects* yang dikelompokkan bersama karena mereka membentuk satu unit konsistensi dan modifikasi. *Aggregates* adalah elemen penting dalam DDD karena mereka memastikan konsistensi dalam data dan perilaku *Domain*.

4. Policies

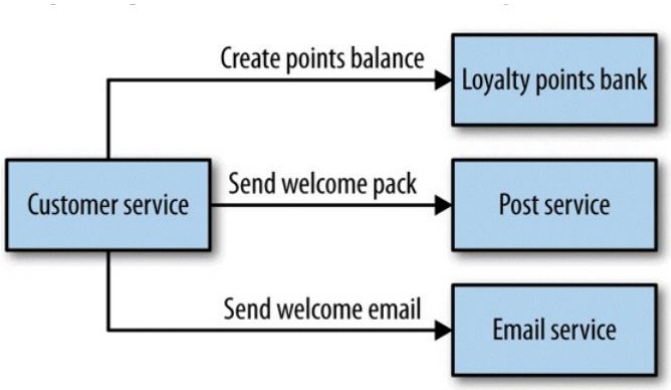
policies adalah aturan atau logika bisnis yang menentukan bagaimana dan kapan *Domain event*s dipicu oleh commands. Mereka dapat mencakup aturan validasi, otorisasi, dan pengambilan keputusan lainnya.

2.9 *Microservices* Communication Pattern

Microservices Communication Pattern merupakan pola atau metode yang digunakan oleh arsitektur *microservices* untuk memungkinkan komunikasi dan kolaborasi antara layanan-layanan kecil dalam sebuah sistem [3]. Pola ini menentukan cara interaksi antar komponen untuk mencapai tujuan bisnis yang diinginkan. Terdapat dua pola umum yang digunakan untuk berkomunikasi antar layanan, yaitu

2.9.1 Orchestration Pattern

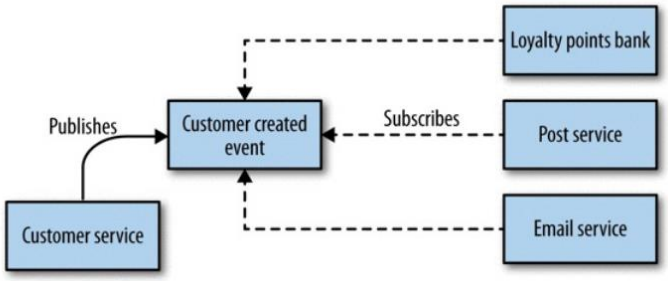
Orchestration Pattern atau *API*-Driven Development sebuah pattern untuk melakukan komunikasi antar *service* dalam artian akan ada satu *service* yang berperan sebagai orkestrator. *Service* orkestrator ini bertanggung terhadap jalannya alur bisnis pada aksi tersebut. Orchestration pattern ini berkomunikasi secara *synchronous,* komunikasi dilakukan via *API call.* Menggunakan *REST*ful, Rpc [11].



Gambar 2.2 Orchestration Pattern

2.9.2 Choreography Pattern

Choreography Pattern atau *Event* Driven *Architecture* (EDA) adalah sebuah pola desain yang digunakan untuk mengatur komunikasi antar layanan (*service*s) dalam arsitektur perangkat lunak terdistribusi [11]. Berbeda dengan pola orchestra, di mana se1mua sistem harus mengetahui secara eksplisit apa yang harus dilakukan ketika suatu peristiwa terjadi, *Event* Driven *Architecture* memungkinkan komunikasi yang *asynchronous* dengan mengirimkan dan menerima peristiwa (*event*s) melalui message broker.



Gambar 2.3 Choreogprahy Pattern

2.10 *API GATEWAY*

Sebagaimana diuraikan sebelumnya, dalam arsitektur Micro*service* akan membagi satu halaman sebuah website kedalam banyak layanan atau *service*, hal tersebut mengakibatkan *client* suatu aplikasi yang berbasis micro*service* akan mengakses suatu layanan secara individual. Oleh sebab itu diperlukan *API* Gateway sebagai titik masuk awal ketika user ingin mengakses suatu layanan. selain itu *API* Gateway juga bertugas untuk menghandle proses seperti management *API*, authentication, Authorization, monitoring dan load balancing [12].

2.11 Advanced Message Queuing Protocol

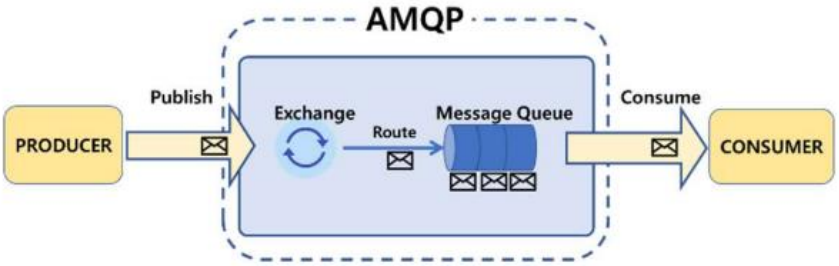
AMQP (Advanced Message Queuing Protocol) adalah protokol standar terbuka yang digunakan untuk sistem messaging, dirancang untuk memastikan pengiriman pesan antara aplikasi atau layanan dalam lingkungan terdistribusi [13]. AMQP memungkinkan aplikasi yang berjalan di platform yang berbeda untuk saling berkomunikasi dan bertukar pesan secara andal, aman, dan efisien.

Beberapa fitur utama AMQP adalah:

1. Interoperabilitas Platform: AMQP memungkinkan aplikasi yang berjalan di berbagai sistem operasi, bahasa pemrog*RAM*an, atau lingkungan runtime untuk berkomunikasi dengan mudah. Protokol ini memungkinkan pertukaran pesan antara aplikasi yang mungkin menggunakan teknologi yang berbeda.
2. *Asynchronous* Messaging: AMQP memungkinkan aplikasi untuk saling berkomunikasi secara *asynchronous*. Ini artinya, pengirim dan penerima pesan tidak perlu aktif atau terhubung pada waktu yang sama. Pesan dapat dikirimkan ke *queue* dan diterima kapan saja oleh aplikasi lain.

3. Reliabilitas dan Durabilitas: AMQP memastikan pesan dapat dikirimkan secara andal bahkan jika salah satu pihak mengalami downtime. Pesan dapat disimpan dalam *queue* hingga dikirimkan dan dikonsumsi oleh penerima, menjaga ketahanan terhadap kegagalan sistem (*fault tolerance*).
4. Flexible Routing: AMQP mendukung berbagai pola routing pesan, seperti *publish/subscribe*, *point-to-point*, dan *fanout*, yang memungkinkan pengembang untuk mendesain komunikasi antar layanan yang fleksibel dan skalabel.
5. Keamanan dan Pengendalian Akses: AMQP menyediakan fitur untuk mengamankan komunikasi dengan menggunakan enkripsi dan otentikasi, serta mengendalikan akses ke pesan dan *queue*.

AMQP biasa digunakan dalam sistem middleware seperti RabbitMQ, dan karena bersifat open standard, ini menjadi pilihan populer bagi perusahaan yang membutuhkan messaging system yang aman, andal, dan mudah diintegrasikan dengan berbagai platform.



2.12 Rabbit MQ

RabbitMQ adalah perangkat lunak message broker yang mengimplementasikan Advanced Message Queuing Protocol (AMQP). Ini berfungsi sebagai perantara untuk mengirim pesan antara aplikasi, memungkinkan aplikasi untuk berkomunikasi dan berinteraksi secara asinkron. RabbitMQ sangat populer dalam arsitektur berbasis mikroservis dan sistem yang memerlukan pertukaran pesan yang handal dan efisien [14] .

1. Producer

Producer adalah aplikasi yang mengirimkan pesan ke RabbitMQ.

2. Consumer

Consumer adalah aplikasi yang menerima pesan dari RabbitMQ.

3. Message

Message adalah paket data yang dikirim antara producer dan consumer. Ini berisi payload dan metadata terkait.

4. Queue

Queue adalah tempat penyimpanan pesan dalam RabbitMQ. Producer mengirim pesan ke queue, dan consumer mengambil pesan dari queue.

2.13 Amazon *Web service*s

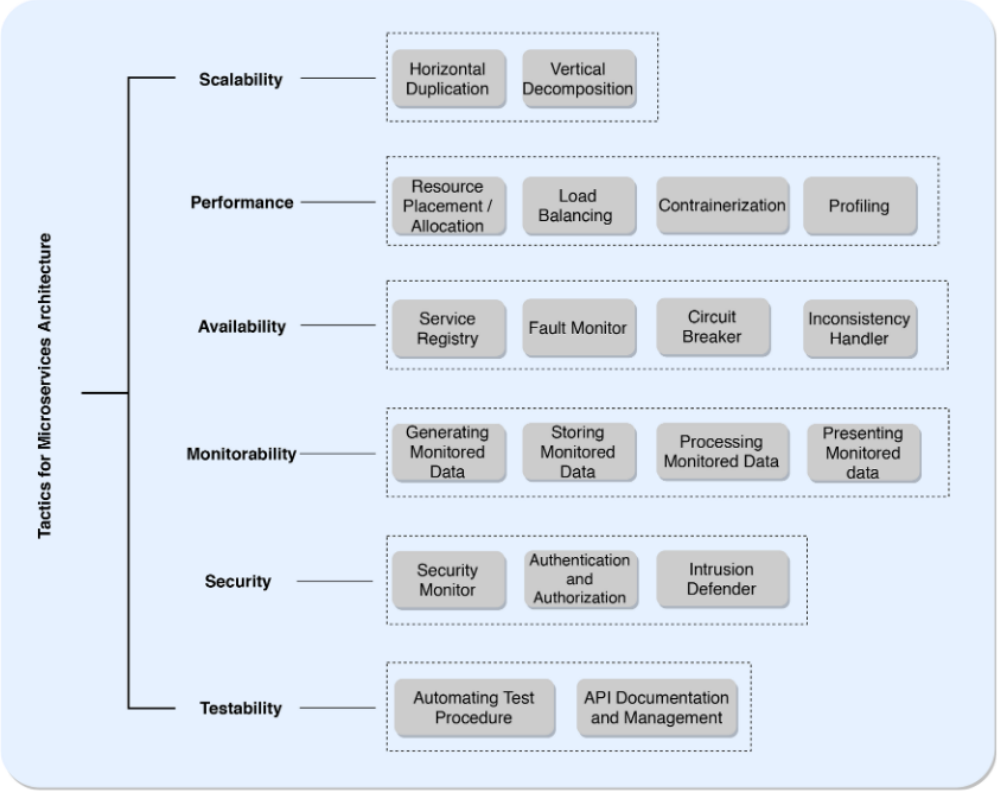
Amazon *Web service*s (AWS) merupakan salah satu penyedia layanan cloud terkemuka yang menawarkan berbagai layanan untuk komputasi, penyimpanan, jaringan, dan analisis data yang dapat diskalakan sesuai kebutuhan pengguna [15]. Dengan demikian, AWS membantu bisnis dan individu memanfaatkan teknologi cloud untuk mencapai efisiensi operasional, inovasi, dan pertumbuhan.

3. Arsitektur teknologi AWS :

AWS menggunakan arsitektur yang terdistribusi dan modular, yang mencakup berbagai layanan utama seperti Amazon Elastic Compute Cloud (EC2) untuk komputasi, Amazon Simple Storage *Service* (S3) untuk penyimpanan data, dan Amazon Relational *Database Service* (RDS) untuk manajemen basis data. Arsitektur ini memungkinkan AWS menyediakan skalabilitas tinggi, di mana pengguna dapat menambah atau mengurangi kapasitas sesuai dengan kebutuhan bisnis. Selain itu, AWS memanfaatkan *regions* dan *availability zones* untuk memastikan ketersediaan layanan yang tinggi dan pengurangan risiko downtime.

2.14 Quality Attributes For *Microservices*

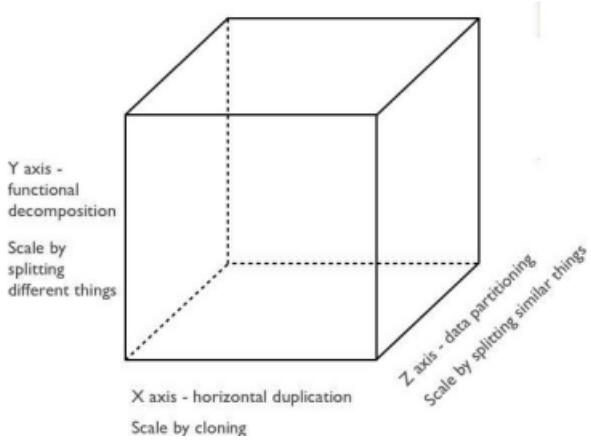
Meskipun *microservices Architecture* (MSA) telah banyak diterapkan, belum ada pandangan yang sistematis mengenai atribut kualitas (*quality attributes*) yang terkait dengan arsitektur ini. Atribut kualitas seperti skalabilitas, performa, ketersediaan, monitorabilitas, keamanan, dan kemampuan uji merupakan perhatian utama dalam MSA [16]. Namun, dampak MSA terhadap atribut-atribut ini serta cara untuk mencapainya masih belum sepenuhnya dipahami [16]. Dari hasil *systematic literature review* dan mengidentifikasi taktik yang dapat digunakan untuk mencapai atribut-atribut kualitas tersebut.



Gambar 2.4 Tactics for *Microservices Architecture*

2.14.1 Scalability

Skalabilitas dapat dilakukan dalam dua arah: horizontal dan vertikal. Skalabilitas horizontal (*scaling out*) dilakukan dengan menambahkan sumber daya ke unit logis, misalnya dengan menambah *server* ke kluster yang sudah ada. Di sisi lain, skalabilitas vertikal (*scaling up*) dilakukan dengan menambahkan lebih banyak sumber daya ke unit fisik, seperti meningkatkan kapasitas memori pada satu komputer. Dalam lingkungan *cloud*, mekanisme skalabilitas horizontal ini sering disebut sebagai *elasticity* [18]. Selain itu, terdapat juga konsep kubus skala yang terdiri dari sumbu X, Y, dan Z untuk menggambarkan pendekatan skalabilitas [19].



Gambar 2.5 The Scale Cube

2.14.1.1 Horizontal duplication (X-Axis Scaling)

Duplikasi horizontal adalah metode skalabilitas di mana beberapa instance dari sebuah aplikasi dijalankan secara paralel di belakang *load balancer*. Jika terdapat N instance, maka setiap instance akan menangani 1/N dari total beban. Pendekatan ini sering digunakan untuk meningkatkan kapasitas aplikasi secara efektif. Setiap instance yang berjalan memiliki akses ke seluruh data, yang dapat meningkatkan kebutuhan memori untuk membuat penggunaan *cache* lebih efisien. Namun, meskipun metode ini meningkatkan kemampuan penanganan beban, duplikasi horizontal tidak secara langsung menyelesaikan tantangan terkait peningkatan kompleksitas pengembangan dan pengelolaan aplikasi, seperti sinkronisasi data antar instance dan manajemen state yang terdistribusi.

2.14.1.2 Vertical decomposition (Y-Axis Scaling)

Dekomposisi vertikal, atau Y-Axis Scaling, adalah teknik pemecahan aplikasi menjadi beberapa layanan independen berdasarkan fungsionalitas yang terkait erat. Setiap layanan bertanggung jawab atas satu atau lebih fungsi spesifik dalam aplikasi. Salah satu metode pemecahan ini adalah *verb-based decomposition*, di mana layanan-layanan diimplementasikan berdasarkan kasus penggunaan tunggal yang logis (kopling logis). Alternatif lain adalah *noun-based decomposition*, di mana layanan-layanan dipecah berdasarkan entitas data tertentu, dan bertanggung jawab atas seluruh operasi terkait entitas tersebut (kopling semantik). Kombinasi dari kedua pendekatan ini juga memungkinkan untuk mencapai fleksibilitas yang lebih tinggi. Dalam penerapan dekomposisi vertikal, granularitas pemecahan layanan harus dipertimbangkan dengan cermat untuk menjaga keseimbangan antara skalabilitas dan performa. Salah satu tantangan utama dalam metode ini adalah memastikan kinerja optimal dari jaringan yang digunakan untuk komunikasi antar layanan.

2.14.1.3 Lookup-Oriented Split (Z-Axis Scaling)

Z-Axis Scaling, atau pemecahan berdasarkan lookup, berfokus pada duplikasi layanan secara horizontal, tetapi dengan pembagian berdasarkan subset data yang berbeda. Dalam pendekatan ini, setiap instance layanan hanya menangani sebagian kecil dari keseluruhan data, bukan seluruh data seperti dalam duplikasi horizontal. Pemecahan ini dilakukan berdasarkan atribut tertentu, seperti ID pelanggan atau wilayah geografis, yang membagi data secara logis untuk memudahkan pencarian (*lookup*) dan mengurangi beban kerja. Dengan cara ini, setiap layanan menangani segmen data yang lebih kecil, sehingga meningkatkan efisiensi dan performa tanpa harus menambah kompleksitas aplikasi secara signifikan. Z-Axis Scaling membantu dalam mengurangi waktu akses data dan mendistribusikan beban kerja secara lebih merata, terutama dalam sistem dengan skala data yang besar.

2.14.2 Performance

Performa adalah pengukuran kemampuan sistem untuk memenuhi persyaratan waktu tertentu sebagai respons terhadap suatu kejadian. Alokasi berbagai jumlah sumber daya dapat memengaruhi throughput atau waktu respons

dari sebuah layanan. Di sisi lain, komunikasi antar *Mikroservices* melalui jaringan dapat secara langsung memengaruhi performa sistem. Dengan demikian, performa dan skalabilitas saling bergantung satu sama lain: *Mikroservice* yang lebih kecil dapat meningkatkan skalabilitas, tetapi dapat mengurangi performa karena jumlah interaksi antar layanan yang lebih tinggi. Sebaliknya, performa mungkin dapat ditingkatkan jika dua Mikro*service*s digabungkan untuk mengurangi overhead komunikasi. Menurut [17], taktik performa dapat dibagi menjadi empat kategori:

2.14.2.1 Resource Management and Allocation

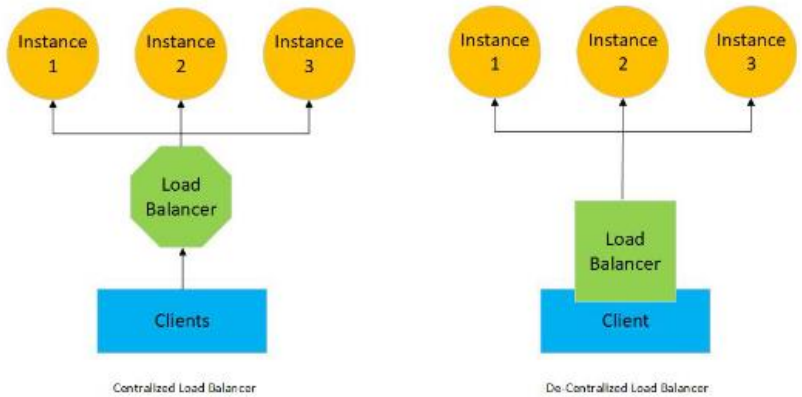
Dalam manajemen sumber daya, upaya dilakukan untuk merespons permintaan dengan cara yang efektif. Permintaan sumber daya sistem tidak dapat dikendalikan sepenuhnya. Oleh karena itu, manajemen efektif dari sumber daya yang tersedia pada sisi respons dapat dianggap sebagai faktor kritis dalam hal performa [18]. Untuk menggunakan taktik ini, *Model* analisis performa biasanya menggunakan analisis "what-if" dan pendekatan perencanaan sumber daya secara sistematis. Sebagai contoh, dalam pe*Model*an performa menyebutkan penggunaan Rantai Markov Kontinu Tiga Dimensi (CMTC) atau Jaringan Petri Stokastik Fluida (FSPN). Tujuan performa yang mungkin mencakup latensi, biaya, atau pemanfaatan sumber daya. Berdasarkan *Model* performa, mesin diperlukan untuk alokasi dan manajemen kontainer, mesin virtual, atau *server* fisik. Kendala utama dalam taktik ini mungkin adalah mencari kombinasi terbaik dari variabel yang relevan dan tujuan spesifik. Untuk menemukan kombinasi ini, simulasi atau eksperimen dapat dilakukan.

2.14.2.2 Load Balancing

Load balancing berusaha untuk mendistribusikan beban masuk dari satu Mikro*service* ke banyak instance. Dengan cara ini, respons terhadap permintaan dapat menjadi lebih cepat dan pemanfaatan kapasitas yang tersedia dapat lebih baik. Selain itu, load balancing memastikan bahwa tidak ada layanan yang terbebani secara signifikan dibandingkan dengan layanan lainnya. Dalam Arsitektur Mikro*service* (MSA), permintaan ke layanan dapat membentuk semacam rantai. Beban tinggi pada layanan tertentu dapat memblokir pemrosesan permintaan dan secara signifikan menurunkan performa. Untuk memastikan distribusi beban yang tepat di antara semua mikro*service*s yang tersedia, digunakan load balancer.

Terdapat dua jenis load balancer untuk mikro*service*s yang dijelaskan dalam sub-bab berikut:

1. *Centralized Load Balancing*: Ini adalah cara yang paling umum digunakan dalam load balancing saat ini. Dengan pendekatan ini, sebuah load balancer terpusat (di sisi *server*) dibuat. Load balancer jenis ini biasanya menggunakan algoritma berbasis push. Dalam pendekatan ini, load balancer terpusat bersifat transparan bagi klien dan klien tidak mengetahui daftar layanan. Load balancer itu sendiri bertanggung jawab atas kumpulan *server* di bawahnya dan memantau kesehatan mereka. Solusi load balancing sisi *server* yang juga disertai dengan. Pendekatan ini memiliki kekurangan ketika load balancer harus menangani sejumlah besar klien dan juga dapat dilihat sebagai titik kegagalan tunggal.
2. *Distributed Load Balancing*: Semakin banyak *Microservices* yang menggunakan pendekatan desentralisasi dalam load balancing. Oleh karena itu, klien itu sendiri memegang load balancer dan menerima daftar *server* yang tersedia, kemudian mendistribusikan permintaan ke beberapa *server* sesuai dengan algoritma load balancing berbasis pull tertentu. Klien memegang daftar *server* yang tersedia dan memilih salah satu dari daftar tersebut baik secara acak atau sesuai dengan algoritma. Netfix mengimplementasikan load balancer sisi klien mereka sendiri, Ribbon, yang juga tersedia sebagai sumber terbuka. Penarikan data yang aktif dan sering oleh klien dapat menyebabkan data yang tidak terbaru setiap saat. Akibatnya, ini dapat menyebabkan pemborosan bandwidth.



2.14.2.3 Containerization

Containerization adalah teknologi yang memungkinkan aplikasi dan layanan dijalankan dalam lingkungan yang terisolasi yang dikenal sebagai kontainer. Berbeda dengan mesin virtual (VM) yang memerlukan sistem operasi tamu lengkap, kontainer berbagi kernel sistem operasi host dan hanya mencakup aplikasi dan dependensinya, yang menghasilkan efisiensi sumber daya yang lebih tinggi dan kecepatan yang lebih baik.

2.14.2.4 Profiling

Profiling adalah teknik yang digunakan untuk mendeteksi masalah kinerja dan mempersiapkan optimasi performa dalam sistem berbasis *microservices*. Teknik ini penting untuk mengidentifikasi dan mengatasi ketidakpastian performa yang muncul di jalur kritis *microservices* [19]. Beberapa keputusan desain arsitektur dapat mempengaruhi kinerja sistem berbasis *microservices*, seperti granularitas layanan dan lingkungan deployment yang dipilih. Profiling membantu dalam mengidentifikasi dampak potensial, bottleneck, serta peluang optimasi dalam sistem. Terdapat dua taktik profiling

1. *CPU* Proling: menganalisis waktu eksekusi layanan dan menemukan titik-titik yang perlu dioptimalkan.

2. *Memory* Proling: menganalisis status memori atau peristiwa alokasi memori. Ini tidak hanya digunakan ketika menemukan kebocoran memori, tetapi juga untuk mengoptimalkan penggunaan memori.

2.15 Docker

Docker adalah platform open-source yang digunakan untuk mengotomatisasi proses pengembangan, pengujian, dan distribusi aplikasi melalui penggunaan container [20]. Container adalah teknologi virtualisasi tingkat aplikasi yang memungkinkan pengembang untuk mengemas aplikasi beserta seluruh dependensi yang dibutuhkannya agar dapat berjalan dengan konsisten di berbagai lingkungan, baik di mesin pengembang, *server*, maupun lingkungan cloud.

Berbeda dengan mesin virtual yang menjalankan seluruh sistem operasi, Docker container berjalan di atas kernel sistem operasi host dan menggunakan l*API*san isolasi yang lebih ringan, sehingga memberikan performa yang lebih baik dan penggunaan sumber daya yang lebih efisien. Docker memungkinkan pengembang untuk membuat lingkungan yang portable dan konsisten, yang dapat meminimalisir masalah kompatibilitas antar lingkungan pengembangan dan produksi.

2.16 Quality of *Service* For *Web service*

Dalam membangun suatu *web service* yang baik dibutuhkan suatu standarisasi sehingga *web service* yang dibangun dapat bekerja dengan baik sesuai dengan tujuannya. Standar ini harus mencakup banyak aspek sehingga *web service* tidak hanya sesuai dengan tujuan, akan tetapi baik dari segi performa, biaya, *maintainability*, dsb. sudah dapat dikatakan optimal. Adapun standar-standar yang harus dipenuhi tersebut meliputi [21] :

a. Performance

Performance dalam *web service* bermaksud kecepatan dalam melayani *request* dari *client* dan memprosesnya. Performance merupakan satu hal yang penting dalam *web service* karena alasan dibangunnya teknologi tidak hanya untuk membantu kehidupan manusia, akan tetapi juga untuk mempercepat pekerjaan manusia. Dengan demikian semakin cepat *web service* melayani dan mengeksekusi *request* dari *client* maka semakin baik pula kinerja *web service* tersebut.

b. Reliability

Reliability merupakan tolak ukur kualitas dari suatu *web service* itu sendiri, *reliability* meliputi tingkat kegagalan dari *web service* dalam melayani *request*. Tingkat kegagalan ini dapat berdasarkan jam, hari, minggu, bulan, atau tahun semua itu kembali kepada pihak developer dalam me-*maintain web service*. Dalam *web service respon* yang dikirim dari *server* tidak ada jaminan akan diterima oleh *client* sesuai dengan *request* yang sudah dikirim. Terkadang *respon* terlambat diterima oleh *client* dan tidak jarang terjadi duplikasi *respon* yang diterima oleh *client*. *Reliability* dapat berarti persentase *web service* sukses melakukan *task* nya dan juga kegagalan yang terjadi.

c. Scalability

Scalability adalah kemampuan *web service* dalam melayani *request* yang datang dengan cara menambahkan sumber daya pada *server* tanpa memberikan dampak pada kinerja. Sehingga *web service* mampu melayani *client* dalam jumlah yang besar sekalipun karena *server* akan otomatis bertambah dan berkurang secara dinamis sesuai kebutuhan. Jika *server* mengalami crash saat beban *server* sedang tinggi, maka dapat disimpulkan jika *server* tidak cukup *scalable* dalam melayani *client*.

d. Accuracy

Accuracy dalam *quality performance* merupakan tingkat kemampuan *service* dalam memberikan *response* untuk *client* sesuai dengan *request* yang terjadi. *Accuracy* akan diukur melalui tingkat kesalahan atau *error* yang terjadi dalam waktu periode tertentu. Dalam hal ini *accuracy* lebih mengukur kepada ketepatan *response* dari *service*, jika hasil yang diukur mendekati satu maka semakin tinggi keakuratannya dan sebaliknya jika hasil justru mendekati nilai nol maka *service* dikatakan tidak akurat.

2.17 *Performance Testing*

Performance Testing adalah metode pengujian yang menilai kinerja aplikasi dengan mempertimbangkan *response time*, *latency*, *throughput*, penggunaan perangkat keras *server*, dan beban kerja. Tujuan utama dari pengujian kinerja adalah untuk mengukur seberapa baik aplikasi berfungsi dalam kondisi uji yang obyektif. Pengujian ini dapat digunakan untuk mengidentifikasi hambatan, menemukan akar penyebab gangguan, serta mengoptimalkan dan meningkatkan konfigurasi platform, baik perangkat keras maupun perangkat lunak, untuk performa yang maksimal. Selain itu, *Performance Testing* juga bertujuan untuk memverifikasi ketahanan aplikasi di bawah tekanan [22].

Karakteristik utama yang diukur dalam *Performance Testing* antara lain:

1. ***Response* time**: Waktu yang dibutuhkan aplikasi untuk merespon permintaan.
2. **Throughput**: Jumlah data yang dapat diproses oleh aplikasi dalam kurun waktu tertentu.
3. **Maksimum pengguna konkuren**: Jumlah maksimum pengguna yang dapat dilayani secara bersamaan oleh aplikasi.

4. **Pemanfaatan sumber daya**: Jumlah *CPU*, *RAM*, jaringan I/O, dan sumber daya disk I/O yang dikonsumsi aplikasi selama pengujian.
5. **Perilaku di bawah pola beban kerja yang berbeda**: Termasuk kondisi beban normal, beban berlebih, dan kondisi di antara keduanya.
6. **Application breaking point**: Kondisi di mana aplikasi berhenti merespon permintaan. Gejala yang umum adalah kesalahan 503 dengan pesan "*Server* Too Busy" dan kesalahan dalam log aplikasi yang menunjukkan kegagalan proses kerja .
7. **Gejala dan penyebab kegagalan**: Terjadi saat aplikasi berada dalam kondisi stres berat.
8. **Titik lemah aplikasi**: Bagian aplikasi yang rentan terhadap kegagalan di bawah beban berat .

2.18 Apache Jmeter

Apache JMeter adalah sebuah alat open-source berbasis Java yang dirancang untuk melakukan pengujian performa dan beban (load testing) pada berbagai jenis *server* aplikasi, seperti *HTTP*, FTP, dan *Database* [23]. JMeter sangat fleksibel dan dapat diperluas melalui *API* yang disediakan. Dalam pengujian aplikasi berbasis web, JMeter sering digunakan untuk menguji responsivitas dan ketahanan *server* terhadap beban yang tinggi.Pengujian menggunakan JMeter biasanya melibatkan pembuatan *Test Plan*, yang merupakan serangkaian operasi yang dijalankan secara otomatis untuk mensimulasikan berbagai skenario penggunaan. Test plan ini dapat disesuaikan dengan berbagai jenis elemen yang ada di dalam JMeter, seperti *Thread Group*, *Samplers*, dan *Listeners*.

1. Thread Group

Thread Group merupakan elemen yang digunakan untuk mengatur jumlah thread (pengguna virtual) yang akan menjalankan pengujian. Setiap thread mensimulasikan satu pengguna, dan dapat dikonfigurasi untuk berjalan dalam jumlah yang besar secara bersamaan, guna mensimulasikan beban *server* pada kondisi penggunaan yang intens.

2. Samplers

Samplers merupakan elemen yang digunakan untuk mengirimkan berbagai jenis *request* ke *server*, seperti *HTTP*, FTP, atau LDAP. Sampler inilah yang mensimulasikan tindakan pengguna, seperti mengirim permintaan *HTTP* GET atau POST. Dalam pengujian web, *HTTP Request* Sampler sering digunakan untuk menguji kinerja layanan web.

3. Listeners

Listeners adalah elemen yang berfungsi untuk memproses hasil dari permintaan yang telah dikirimkan oleh Sampler. Listeners dapat digunakan untuk menyimpan hasil pengujian ke dalam file atau menampilkan hasil tersebut dalam bentuk grafik, tabel, atau laporan statistik. Beberapa listener seperti Summary Report dan Graph Results memudahkan dalam melakukan analisis hasil pengujian.