BAB 2
LANDASAN TEORI

2.1 Web Service

Web Service merupakan salah satu komponen sistem terdistribusi pada *SOA* (*Service Oriented Architecture*). *Service* dibangun untuk menyediakan *interface* yang memproses dan mengirim pesan dalam bentuk *xml* atau *json*. *Service* menggunakan *object oriented* untuk mendefinisikan pesan yang akan dikirim kepada *client* atau *Service* yang lain. Selain menyediakan *interface* untuk menerima dan mengirim pesan dari *client* atau *Service* lain melalui protokol (*HTTP, SMTP, TCP*), *Service* juga menangani beberapa *business logic* dan beberapa operasi *CRUD* di dalamnya.

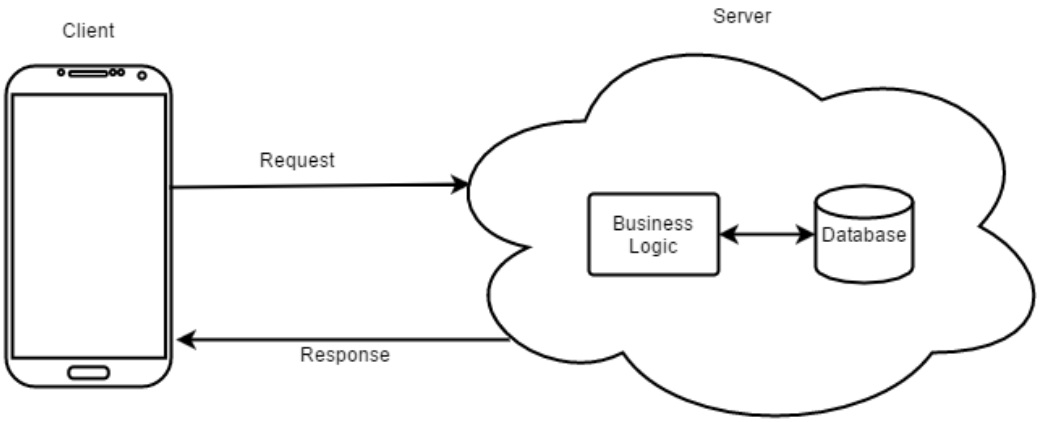
2.2 Quality of Service for Web Service

Untuk membangun sebuah sistem informasi yang berukuran kecil hingga besar diperlukan sebuah standarisasi dan mekanisme, sehingga sistem yang dibangun memiliki kualitas yang baik, selain itu mudah untuk *maintenance*. Untuk membangun sebuah sistem informasi *Web Service* ada beberapa standar yang harus diperhatikan di antaranya. [5]

1. *Performance*

Performance merupakan salah satu hal yang sangat penting dalam dunia teknologi karena tujuan dibangunnya sebuah teknologi diharapkan dapat membantu pekerjaan manusia, namun tidak hanya sekedar membantu tapi mempercepat pekerjaan, sebagai contoh dalam hal menghitung, kalkulator dapat membantu pekerjaan menghitung dalam waktu yang jauh lebih cepat. Oleh karena itu dalam pembangunan sistem, *performance* perlu perhatian khusus, mulai dari algoritma yang digunakan, *stack*, hingga *hardware* yang akan menjalankan sistem yang dibangun.

Pengukuran *performance* pada *QoS Web Service* lebih fokus pada pengukuran throughput, response time, dan latency [6] [7] [8]. Throughput merepresentasikan permintaan/satuan waktu, dimana throughput ini mengukur seberapa banyak request yang ditangani atau di-handle oleh *server* secara periodik. Response time mengukur berapa lama *server* mengirimkan data kepada client. Latency mengukur waktu delay antara waktu client melakukan request dan waktu ketika client mendapatkan response dari *server*.



Gambar 2-1 Client *Server* Transaction

1. Throughput

Pengukuran throughput dapat dilakukan dengan cara melakukan lebih dari satu request dalam kurun waktu tertentu dan dalam dilakukan secara bersamaan. Untuk mendapatkan nilai maksimal throughput dapat menggunakan rumus sebagai berikut:

$$\text{Maksimal throughput} = \frac{jumlah\ maksimal\ request\ ditangani}{satuan\ waktu} \ldots \text{persamaan}(2.1)$$

Proses pengukuran nilai maksimal throughput dapat dilakukan dengan melakukan logging atau pencatatan jumlah request yang di-handle oleh *server* dalam kurun waktu tertentu.

2. Latency

Pengukuran waktu latency merupakan waktu delay dimana client menunggu response yang diberikan oleh *server*. Untuk mendapatkan nilai waktu delay dapat menggunakan rumus sebagai berikut:

Latency = response time – request time .....persamaan(2.2)

Pengukuran latency dapat dilakukan dengan cara melakukakan logging atau pencatatan pada client, yang dapat dicatat adalah waktu ketika client melakukan request ke *server* dan waktu ketika client mendapatkan response dari sever. Setelah waktu tersebut didapatkan, maka rumus diatas dapat digunakan untuk mendapatkan nilai latency.

3. Response Time

Pada pengukuran ini merupakan perhitungan waktu yang diperlukan untuk menyelesaikan request. Waktu yang dihitung adalah waktu dimana *server* melakukan execution time dan memberikan response kepada client. Untuk mendapatkan nilai response time dapat menggunakan rumus sebagai berikut:

Response time = execution time + data transferred time ..... persamaan(2.3)

Data transfer time merupakan waktu dimana *server* mengirimkan data json ke client. Pengukuran ini juga dapat menggunakan logging atau pencatatan, waktu yang dicatat adalah waktu yang diperlukan untuk business logic yang didapat dari waktu ketika *server* mendapatkan request hingga business logic selesai dilakukan, dan waktu yang diperlukan untuk mengirim data ke client yang didapat dari waktu business logic selesai dilakukan hingga data tersebut sampai di client.

Parameter qualitas *performance* pada *Web Service* dapat dilihat dari empat point yang telah dijabarkan sebelumnya yakni throughput, response time, latency

dan execution time. *Web Service* yang memiliki *performance* yang baik dapat diliha dari throughput yang cepat, latency yang rendah, response yang cepat, dan execution cepat.

2. Reliability

Reliability merupakan tolak ukur untuk tingkat maintain *Web Service*. Mengukur tingkat kegagalan pada sistem setiap hari, minggu, bulan, atau tahun tergantung dari seberapa besar kemampuan pengembang untuk melakukan maintain. Karena tidak ada jaminan semua pesan atau response akan sampai ke tujuan.

3. Scalability

Dalam pembangunan sebuah sistem akan dimulai dari *prototype* di mana sistem belum siap untuk digunakan secara utuh, karena beberapa fungsi masih memiliki *error* atau *bug*. Setelah semuanya fungsional dapat dijalankan sesuai dengan harapan *developer* tidak akan berhenti untuk mengembangkan lagi sistem yang telah dibangun, oleh karena itu sistem sangat dituntut untuk dapat dikembangkan menjadi sistem yang sesuai dengan kebutuhan saat itu. Misalnya dalam meningkatkan kemampuan sistem dalam memproses setiap *request* dari *user* yang semakin meningkat jumlahnya.

4. Capacity

Web Service saat ini salah satu *distributed system* yang sering digunakan untuk transaksi data, *client* atau *system* yang menggunakan layanan dari *Web Service* tersebut bermacam-macam dari *Web , desktop application,* hingga *mobile*. Pada kasus *m-commerce* saat ini sudah banyak aplikasi yang menyediakan layanan untuk membeli barang secara *online* atau pengguna dapat menjadi penjual untuk pengguna lainnya. Pengguna pada aplikasi lazada telah mencapai lebih dari sepuluh juta *download*, dengan demikian *Web Service* lazada harus mampu untuk meng-*handle request* dari pengguna yang begitu banyak dalam waktu yang bersamaan.

5. Robustness

Request client yang memiliki parameter untuk diproses di dalam *server* yang terkadang ada beberapa parameter tidak memiliki nilai yang berakibat kesalahan pada sistem yang dikarenakan *human error* dari pengguna atau validasi pada sistem masih kurang dan sistem *Web Service* dituntut untuk dapat menangani masalah tersebut.

6. Exception Handling

Dalam pembangunan sistem memungkinkan beberapa hal yang tidak terprediksi dapat terjadi. Contoh sederhana dalam hal *request*, *client* akan melakukan *request* ke *server* dengan beberapa parameter, namun tidak semua parameter bersifat *required*, maka *server* harus mampu untuk menanggulangi jika ada parameter yang tidak bersifat *required* tidak memiliki nilai.

7. Accuracy

Akurasi merupakan hal penting dalam sistem informasi, namun di sini maksud dari akurasi adalah tingkat kesalahan yang dihasilkan oleh layanan *Web* .

8. Integrity

Integrity salah satu hal yang wajib ada di dalam *Web Service* untuk mencegah *unauthorized access* atau modifikasi pada sistem atau data. Ada dua tipe *integrity* yakni data *integrity* dan *transactional integrity*. Data *integrity* memastikan apakah data yang di transfer telah dimodifikasi ketika transit. Sedangkan *transactional integrity* mengacu pada prosedur yang menjamin menjaga *database integrity* dalam transaksi.

9. Accessibility

Client dari *Web Service* tidak hanya satu jenis, ada beberapa jenis *client* yakni *mobile, desktop*, maupun *Service* lainnya. Jadi *Web Service* harus mampu untuk melayani setiap *request* yang berbeda-beda.

10. Availability

Pembangunan *Web Service* api tidak semata-mata dibangun dari awal sebuah startup atau perusahaan dari awal, terkadang pembangunan *Web Service* dibangun di tengah jalan dengan alasan permintaan dari pengguna atau ekspansi dari perusahaan itu sendiri ataupun rencana untuk membangun sistem terintegrasi. Maka dari itu *Web Service* harus mampu tersedia sesegera mungkin untuk berbagai keperluan dalam integrase ataupun ekspansi.

11. Interoperability

Dalam pembangunan sebuah sistem informasi khususnya *Web Service* tidak hanya memikirkan masalah bahasa pemrograman yang digunakan namun ada beberapa hal seperti sistem dari *database*. Tidak hanya itu developer juga harus memikirkan bagaimana untuk menggunakan sistem dari *database* di dalam *Web Service* yang dibangun. Selain *database* ada juga sistem lain yakni jalur komunikasi seperti *http protocol*. Jadi *Web Service* harus memiliki kemampuan untuk berinteraksi dengan sistem lain tanpa akses yang rumit.

12. Security

Salah satu media sosial dunia yakni *facebook* memiliki jutaan data pribadi mengenai penggunanya di seluruh dunia. Data tersebut bersifat rahasia, tidak sembarang orang atau kelompok dapat mengakses data tersebut. Sehingga perusahaan *facebook* harus mampu untuk menjaga semua data dari setiap penggunanya. Kini *facebook* telah menyediakan *Service* untuk perusahaan atau aplikasi lain agar dapat mengakses akun tertentu dengan beberapa kebijakan dari *facebook* untuk menjaga data pengguna agar tidak semua data dapat diambil begitu saja. Maka dari itu *Web Service* yang dibangun harus memiliki sistem keamanan seperti *authentication, authorization, confidentiality, traceability and auditability, data encryption, non-repudation.*

2.3 GraphQL

GraphQL maupun *REST* tidak terpaku pada *database* yang digunakan, pada dasarnya *Web Service* mengambil sejumlah data dari *database* yang nantinya akan

di proyeksikan ke dalam bentuk API. Prinsip-prinsip yang digunakan *GraphQL* untuk membuat suatu *interface* atau API sebagai berikut: [9]

1. Hierarchical

GraphQL menggunakan hierarki untuk relasi antar objek, seperti halnya dalam REST menggunakan *join* statement pada SQL.

2. Client Specified Queries

Pada *GraphQL* setiap *client request* untuk *data fetching* atau *data manupulate* menggunakan *query* yang di dalamnya berisi model data yang dibutuhkan, berbeda dengan REST, semuanya ketentuan data yang akan disajikan dalam API sudah ditentukan, karena konsep *everything is resource* pada REST.

3. Backward Compabilities

Setiap *mobile application* minimal dalam tiga bulan akan diperbaharui, pada kasus ini setiap melakukan *request* data ada kemungkinan berbeda-beda setiap *client*, karena tidak semua *client* akan memperbaharui aplikasinya. sehingga *GraphQL* dari itu setiap melakukan *requent* data ke *GraphQL* menggunakan *specified query* bukan *resource.*

4. Strongly Typed

Ketika *client* mengirim *request* dengan *query*, setiap *item* yang ada di dalam objek *query* akan di cek satu per satu apakah data yang di-*request* tersedia atau tidak, jika tidak tersedia maka *GraphQL* akan membuat pesan *error* sebelum *query* di eksekusi.

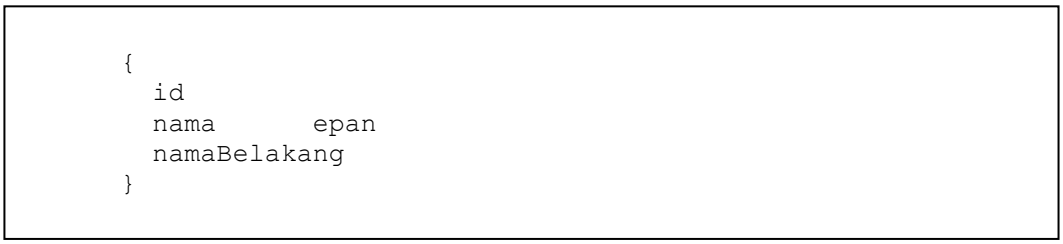
GraphQL dibangun dalam waktu yang cukup lama, karena pembangunan *GraphQL* ini diharapkan dapat menyelesaikan beberapa permasalahan pada kasus tertentu. [10]

1. Operation

Teknologi *GraphQL* memiliki dua buah operasi yang akan digunakan sebagai alat untuk transaksi data *client-server* yakni *query* dan *mutation*, *query* hanya digunakan untuk *fetching* data sedangkan *mutation* digunakan untuk modifikasi data.

2. Selection Set

Selection set merupakan sebuah set informasi yang dibutuhkan oleh client, dalam *selection set* dapat berisi *Field*, *FragmentSpread*, atau *InlineFragment*. Dan *client* akan mendapatkan data yang sama persis dengan *selection set* yang dikirim ke *server*. Berikut contoh bentuk dari *selection set*.



Gambar 2-2 Selection Set

3. Field

Sebuah *selection set* terbentuk dari satu atau beberapa *field*. Sebuah *field* menjelaskan satu informasi. Kumpulan *field* dalam satu *selection set* menjelaskan data yang kompleks atau keterhubungan dengan data yang lain. Untuk mendapatkan data yang lebih kompleks dalam data, memungkinkan dalam sebuah *field* berisi *selection set*. Semua operasi pada *GraphQL* harus menentukan *selections* pada *field* yang mengembalikan nilai *scalar* untuk memastikan *response* yang didapat jelas.

```
{
  me {
    id
    firstName
    lastName
    birthday {
      month
      day
    }
    friends {
      name
    }
  }
}
```

Gambar 2-3 Field

Field di level pertama pada *selection* set menandakan informasi yang dapat diakses secara global dari *client* dan informasi tersebut merupakan informasi *client* utama atau *current viewer*.

4. Arguments

Secara konsep *field* merupakan sebuah *functions* yang mengembalikan nilai, dan dalam beberapa kasus *field* menerima *argument* yang dapat mengubah nilai dari hasil *request*. Sebagai contoh *client* akan me-*request* data *user* dengan *id* 4 dengan spesifikasi *profile picture* dengan ukuran 100 x 100,

```
{
  user(id: 4) {
    id
    nama
    profilePic(size: 100)
  }
}
```

Gambar 2-4 Arguments

5. Field Alias

Secara *default* objek dari *response* akan menggunakan nama *field*. Namun nama *field* yang digunakan juga diganti dengan nama alias sesuai dengan kebutuhan, nama alias digunakan untuk mempermudah dalam pembacaan data *response* dan menghindari duplikat *key*. Berikut contoh dari penggunaan alias.

```
{
  user(id: 4) {
    id
    name
    smallPic: profilePic(size: 64)
    bigPic: profilePic(size: 1024)
  }
}
```

Gambar 2-5 Field Alias

Dari objek *request* tersebut maka *response* dari *server* akan berbentuk seperti berikut.

```
{
  "user": {
    "id": 4,
    "name": "Mark Zuckerberg",
    "smallPic": "https://cdn.site.io/pic-4-64.jpg",
    "bigPic": "https://cdn.site.io/pic-4-1024.jpg"
  }
}
```

Gambar 2-6 Result Field Alias

6. Fragments

Fragment merupakan unit utama dalam komposisi *GraphQL*. *Fragment* memungkinkan untuk menggunakan *selections* berulang kali pada sebuah *field*. *Inline fragment* dapat digunakan di dalam *selection* . . . . . .

```
query noFragments {
  user(id: 4) {
    friends(first: 10) {
      id
      name
      profilePic(size: 50)
    }

    mutualFriends(first: 10) {
      id
      name
      profilePic(size: 50)
    }
  }
}
```

Gambar 2-7 Without Fragment

Selection set friends dan *mutualFriend* memiliki *field* yang sama, maka *selection set* ini dapat dimasukkan ke dalam sebuah *fragment*.

```
query withFragments {
  user(id: 4) {
    friends(first: 10) {
      . . .friendFields
    }

    mutualFriends(first: 10) {
      . . .friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  profilePic(size: 50)
}
```

Gambar 2-8 With Fragment

Memakai *fragment* menggunakan operator (. . .). Semua *field* yang berada di dalam *fragment* akan dimasukkan ke dalam *field query* yang menggunakan *fragment*. *Fragment* juga dapat diaplikasikan pada *multiple levels fragment*, berikut contoh dari *multiple level*.

```
query withNestedFragments {
  user(id: 4) {
    friends(first: 10) {
      . . .friendFields
    }

    mutualFriends(first: 10) {
      . . .friendFields
    }
  }
}

fragment friendFields on User {
  id
  name
  . . .standardProfilePic
}

fragment standardProfilePic on User {
  profilePic(size: 50)
}
```

Gambar 2-9 Profile Picture Fragment

2.4 REST

REST mendefinisikan satu set prinsip arsitektur di mana dapat digunakan untuk mendisain *Web Service* yang berfokus pada *resource* sistem, termasuk bagaimana *resource* memiliki alamat (*URL address*) dan dapat dikirim kepada *client* yang memiliki bahasa yang berbeda-beda. Interaksi antara *client* dengan *server* menggunakan *interface* yang bisa diakses oleh protokol *http* yang sering disebut sebagai API (*Application Programming Interface*). Dikarenakan pembangunan *Service* untuk interaksi atau manipulasi data menggunakan bahasa *native* dari SQL *database* yaitu *insert*, *select*, *update*, *delete*, maka *verb* yang digunakan ada empat di antaranya *post*, *get*, *put*, dan *delete*. Dalam pembangunan *RESTful Web Service api's* ada beberapa prinsip yang digunakan, di antaranya: [11]

1. Each resource is identifiable by a unique

Jumlah *resource* pada suatu sistem lebih dari satu, dan untuk mengakses *resource* tersebut diperlukan *URI*, namun *URI* harus bersifat unik sama halnya

dengan nomor ktp, satu sama lain harus berbeda untuk mengidentifikasi pemilik dari nomor ktp tersebut.

2. User the standard HTTP methods

   Http protocol memiliki delapan *verb* yaitu sebagai berikut:

   1. GET
   2. POST
   3. PUT
   4. DELETE
   5. HEAD
   6. OPTIONS
   7. TRACE
   8. CONNECT

Pada teknologi *RESTful* ini tidak semua *verb* HTTP digunakan, hanya empat yang digunakan di antaranya

Tabel 2-1 Verb *RESTful*

HTTP Verb	**Action**	**Kode status respon**
GET	*Request* terhadap *resource* yang sudah ada	**"200 OK",** jika *resource* tersedia, **"404 Not Found"**, jika *resource* tidak tersedia, dan **"500 Intenal *Server* Error",** jika terdapat kesalahan di dalam *server*.
PUT	Memasukan data atau memperbaharui data	**"201 CREATED",** jika data telah masuk ke dalam *database*, **"200 OK"**, jika data telah diperbaharui, **"500 Intenal *Server* Error",** jika terdapat kesalahan di dalam *server*.
POST	Memperbaharui data	**"200 OK"**, jika data telah diperbaharui, **"500 Intenal *Server* Error",** jika terdapat kesalahan di dalam *server*.
DELETE	Menghapus data	**"200 OK",** jika *resource* terhapus, **"404 Not Found"**, jika *resource* yang akan dihapus tidak tersedia, dan **"500 Intenal *Server* Error",** jika terdapat kesalahan di dalam *server*.

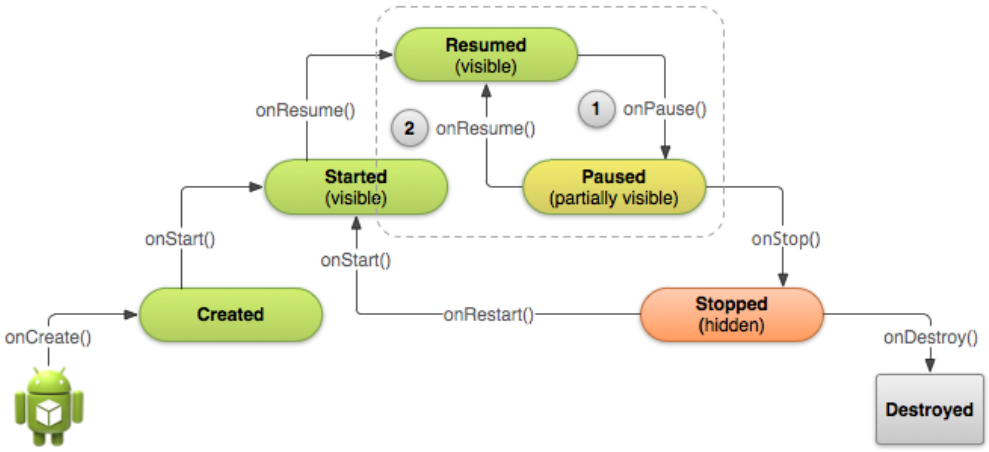
3. Resource can have multiple representations

REST dapat mendukung beberapa format data representasi tergantung pada *request* yang dilakukan, selama *server* mendukung format tersebut.

2.5 Android

Android merupakan sebuah sistem operasi yang berjalan dia atas *linux karnel*, sistem operasi ini diterapkan pada *mobile device* seperti *smartphone*, komputer tablet, dan *smartwatch*, tidak hanya *mobile device android* telah digunakan pada kendaraan roda empat dan *television*. *Android* pertama kali dikembangkan oleh Android Inc., dan pada tahun 2005 Google membeli dan mengembangkannya hingga saat ini.

Android memiliki paradigma pemrograman yang berbeda dengan pemrograman pada umumnya dimana dalam pemrograman *java*, *method* yang pertama kali dijalankan adalah main(), sedangkan pada *android* menjalankan kode dalam *method Activity* dengan menerapkan *callback* tertentu yang sesuai dengan tahap tertentu dari siklus hidup. siklus hidup berguna untuk memastikan aplikasi tidak menghabiskan sumber daya baterai.



Gambar 2-10 Android Lifecycle

Terdapat beberapa state dalam siklus hidup android yang terjadi seperti diilustrasikan pada Gambar 2-10 Android Lifecycle

1. Created

   Sebuah aplikasi pertama kali dijalankan dan mengeksekusi *method onCreate*() termasuk menginisialisasi komponen tampilan.
2. Started

   State ini secara langsung akan diakses setelah *onCreate*() dijalankan, dan ketika aplikasi dijalankan setelah aplikasi tersebut di tutup.
3. Resumed

   State *resumed* akan dijalankan setelah *state paused*. *state* ini akan menjalankan perintah yang ada di dalam *method onResume()*..
4. Paused

   Pada *state* ini semua aktivitas dalam aplikasi akan dihentikan sementara. *state* ini berjalan ketika ada aplikasi yang berjalan dengan prioritas lebih tinggi. aplikasi tidak bisa menerima *input* dari *user* dan tidak bisa mengeksekusi perintah kode program.
5. Stopped

   Pada *state* ini aplikasi dalam keadaan mati, tetapi *background Service* tetap berjalan.

6. Destroyed

   Aplikasi benar - benar mati, tidak ada proses apapun yang berjalan termasuk *background Service*.

2.6 BlazeMeter

Pengujian *performance* menggunakan acuan *QoS Web Service* dapat dilakukan dengan menggunakan salah satu tool BlazeMeter. Tool BlazeMeter berbasis cloud untuk mempermudah pengujian. BlazeMeter memiliki fitur yang dibutuhkan untuk menguji *performance* di antaranya fitur response time, thoughput, dan delay. Dalam menguji *performance* BlazeMeter mengeksekusi client dan *server* pada cloud, sehingga ketika melakukan pengujian BlazeMeter dapat ditutup atau melakukan pengujian yang lainya. Selain itu BlazeMeter juga dapat dikonfigurasi sesuai dengan kebutuhan mulai dari jumlah user virtual yang

akan digunakan untuk mengakses *Service* api, jumlah iteration untuk request, waktu yang digunakan untuk menguji *Web Service* api, dan jumlah waktu delay antar requet untuk satu user virtual. Hasil pengujian disajikan dengan grafik dan data numerik untuk setiap mengujian secara detail.

2.7 Heroku cloud

Heroku merupakan salah satu penyedia *platform-as-a-Service* (PaaS) terkemuka di bisnis *cloud software*, yang membuktikan dirinya sebagai solusi PaaS terkemuka untuk perusahaan kecil dan besar. Dengan peningkatan yang konsisten dan filosofinya kenyamanan melebihi konfigurasi, menjadikan developer untuk fokus dalam menulis aplikasi *Web* . Hal yang bersangkutan dengan *server*, *building*, *deploying*, *running*, dan *scaling* aplikasi akan diatasi oleh Heroku.

Heroku menyediakan dukungan platform untuk Ruby, Ruby on Rails, Java, Node.js, Clojure, Scala, Python, dan PHP. Arsitektur Heroku add-on memungkinkan developer menyesuaikan penggunaan berbagai paket pihak ketiga berdasarkan kebutuhan. Developer memiliki fleksibilitas untuk memilih *plan basic* atau *premium* berdasarkan kebutuhan *Web* site. Developer juga bisa mempercepat aplikasi dengan menambahkan *resources* seperti memcached untuk *caching* data, sehingga menciptakan aplikasi yang *powerful* yang kaya dengan fitur [12].

2.8 JSON

Interaksi antara *client* dengan *server* mempunyai banyak cara, salah satu cara yang cukup familiar adalah interaksi yang menggunakan *interface* antara satu sistem dengan sistem lainnya yang memiliki platform yang berbeda ataupun sama, atau sering disebut sebagai API(*Application Promgramming Interface*). Sebuah interaksi memiliki min satu pesan *didalamnya*, format pesan yang sering digunakan dalam *interaksai client-server* adalah *xml* dan *json*. *Json* merupakan salah satu format tulisan yang ringan, mudah dibaca oleh manusia ataupun mesin, dan mudah bagi mesin untuk men-*genterate* atau untuk memecahkan atau mengubah menjadi bentuk lain sesuai dengan kebutuhan dari satu pesan *json*. [13]

2.9 M-Commerce

Perkembangan dalam bidang penjualan semakin meningkat sejak hadirnya penjualan *online* atau dapat disebut sebagai *e-commerce*. Banyak kalangan memulai usaha dengan mengandalalkan *e-commerce*, karena tidak perlu membeli atau menyewa bangunan untuk sebuah toko, cukup dengan membuat sistem penjualan *online* atau *e-commerce* penjual cukup diam di rumah dan sediakan satu ruangan sebagai gudang. *E-comnmerce* memiliki beberapa jenis berdasarkan peran di dalamnya, yaitu B2B (*Business to Business*), B2C (*Business to Client*), C2C (*Clinet to Client*).

2.10 Object Oriented Analysis And Design

Analisis dan desain berorientasi objek adalah cara baru dalam memikirkan satu masalah dengan menggunakan model yang dibuat menurut konsep sekitar dunia nyata. Tujuan dari analisis berorientasi objek adalah untuk mengembangkan model yang menggambarkan perangkat lunak komputer karena bekerja untuk memenuhi seperangkat persyaratan yang ditentukan *usir* [14]. *Tools* yang dapat digunakan pada pendekatan analisis pengembangan sistem secara objek yaitu menggunakan UML.

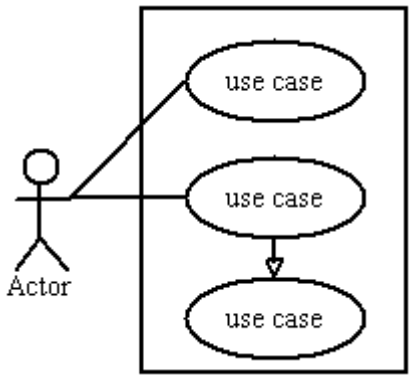
Unified Modelling Language (UML) adalah sebuah bahasa yang telah menjadi standar dalam industri untuk visualisasi, merancang dan mendokumentasikan sistem piranti lunak. UML menggunakan *class* dan *operation object* dalam konsep dasarnya, maka ia lebih cocok untuk penulisan piranti lunak dalam bahasa bahasa berorientasi objek [15]. Dalam membangun block UML ada tiga hal yang harus diperhatikan, yaitu *object* (memodelkan konsep), *relationship* (menghubungkan *object*), dan *diagram* (*grouping* yang saling menghubungkan antara *object* dan *relationship*. Diagram yang umum dipakai dalam analisis dan desain adalah:

1. *Use Case Diagram*

Use Case Diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah “apa” yang diperbuat sistem, dan bukan “bagaimana”. Sebuah *Use case* merepresentasikan sebuah interaksi antara aktor

dengan sistem. Sebuah *use case* dapat meng-*include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa use case yang di-*include* akan dipanggil setiap kali use case yang meng-*include* dieksekusi secara normal. Sebuah use case dapat di-*include* oleh lebih dari satu use case lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang serupa. Sebuah use case juga dapat meng-*extend usecase* lain dengan *behaviour*-nya sendiri. Sementara hubungan generalisasi antar *use case* menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain.

Dasar menentukan sebuah *use case* adalah *use case* merupakan sesuatu yang menyediakan beberapa hasil terukur kepada pengguna atau sistem eksternal. *Use case* harus memiliki sangat jelas kriteria lulus / gagal. Pengembang, tester, penulis teknis, dan pengguna harus secara eksplisit tahu apakah sistem memenuhi kasus penggunaan atau tidak. Setiap bagian dari *use case* yang memenuhi tes sederhana ini mungkin menjadi kandidat yang baik untuk *use Chase.* Dasar pembangunan *use case diagram* dapat dilihat pada Gambar 2-11 Dasar Pembangunan Use Case Diagram.



Gambar 2-11 Dasar Pembangunan Use Case Diagram

2. *Use Case Scenario*

Sebuah diagram yang menunjukkan *use case* dan aktor mungkin menjadi titik awal yang bagus, tetapi tidak memberikan detail yang cukup untuk desainer sistem untuk benar-benar memahami persis bagaimana sistem dapat terpenuhi. Cara terbaik untuk mengungkapkan informasi penting ini adalah dalam bentuk penggunaan *use case scenario* berbasis teks per u*se case*nya. Berikut adalah dasar

format penulisan *use case scenario* [16]. Dasar pembangunan *use case scenario* dapat dilihat pada **Error! Reference source not found.**.

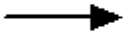
Tabel 2-2 Dasar Pembangunan Use Case Scenario

Use Case Name	Berisi nama dari *use case* yang akan digunakan	
Goal In Context	Menjelaskan apa yang aktor coba untuk dapatkan dari *use case*	
Description	Menjelaskan gambaran dari *use case*	
Related Use Case	Daftar *use case* yang berhubungan dengan *use case* tersebut	
Successful End Condition	Kondisi *use case* jika berhasil	
Failed End Condition	Kondisi *use case* jika gagal	
Actors	Daftar aktor yang dapat mengakses *use case*	
Trigger	Aktifitas yang dilakukan untuk mengawali *use case*	
Main Flow	Step	Action
	1	Deskripsi urutan aksi dari aktifitas *use case*
	2	
Extension	Step	Branching Action
	2.1	Deskripsi urutan aksi lain selain urutan aksi utama

3. *Sequence Diagram*

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait). *Sequence diagram* biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari apa yang men-*trigger* aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara internal dan *output* apa yang dihasilkan.

Masing-masing objek, termasuk aktor, memiliki *lifeline* vertikal. *Message* digambarkan sebagai garis berpanah dari satu objek ke objek lainnya. Pada fase desain berikutnya, *message* akan dipetakan menjadi operasi/metoda dari *class*. *Activation bar* menunjukkan lamanya eksekusi sebuah proses, biasanya diawali dengan diterimanya sebuah *message* [15]. Untuk objek-objek yang memiliki sifat khusus, standar UML mendefinisikan icon khusus untuk objek *boundary*, *controller* dan *persistent entity*. Dasar pembangunan *sequence diagram* dapat dilihat pada Gambar 2-12 Dasar Pembangunan Sequence Diagram.

Arrow	Message type
	Simple
	Synchronous
	Asynchronous
	Balking
	Time out

Gambar 2-12 Dasar Pembangunan Sequence Diagram

4. *Class Diagram*

Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. Class menggambarkan keadaan (atribut/properti) satu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi). Class diagram menggambarkan struktur dan deskripsi *class*, *package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain. Class memiliki tiga area pokok:

a. Nama dan stereotype
b. Atribut
c. Metoda

Atribut dan metoda dapat memiliki salah satu sifat berikut:

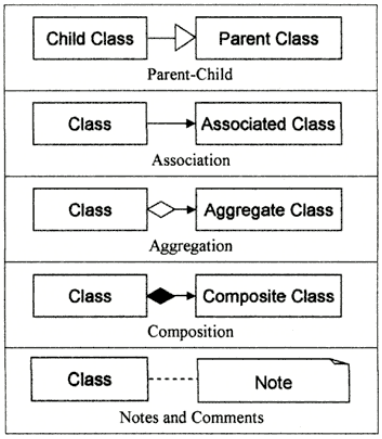
a. Private, tidak dapat dipanggil dari luar class yang bersangkutan.
b. Protected, hanya dapat dipanggil oleh class yang bersangkutan dan anak-anak yang mewarisinya.
c. Public, dapat dipanggil oleh siapa saja.

Class dapat merupakan implementasi dari sebuah interface, yaitu class abstrak yang hanya memiliki metoda. Interface tidak dapat langsung diinstansiasikan, tetapi harus diimplementasikan dahulu menjadi sebuah class.

Sesuai dengan perkembangan class model, class dapat dikelompokkan menjadi package. Kita juga dapat membuat diagram yang terdiri atas package.

Class memiliki tipe-tipe relationship, diantaranya:

a. Asosiasi, yaitu hubungan statis antar class. Umumnya menggambarkan class yang memiliki atribut berupa class lain, atau class yang harus mengetahui eksistensi class lain.
b. Agregasi, yaitu hubungan yang menyatakan bagian terdiri atas dimana ketika satu class di share atau direferensikan kepada objek yang ada di class lain.
c. Pewarisan, yaitu hubungan hirarkis antar class. Class dapat diturunkan dari class lain dan mewarisi semua atribut dan metoda class asalnya dan menambahkan fungsionalitas baru, sehingga ia disebut anak dari class yang diwarisinya. Kebalikan dari pewarisan adalah generalisasi.
d. Komposisi, yaitu jenis relasi class diagram yang kuat dimana jika sebuah class tidak bisa berdiri sendiri dan harus merupakan bagian dari class yang lain, maka class tersebut memiliki relasi Composition terhadap class tempat dia bergantung tersebut. Sebuah relationship composition digambarkan sebagai garis dengan ujung berbentuk jajaran genjang berisi/solid.
e. Depedensi, salah satu jenis relasi class diagram yang lemah dimana objek dalam suatu class akan bekerja sangat singkat dengan objek yang ada pada class lain.



Gambar 2-13 Dasar Pembangunan Class Diagram

5. *Activity Diagram*

Activity diagram menggambarkan berbagai alur aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing aliran berawal, *decision* yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity* diagram juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi. Sebuah aktivitas dapat direalisasikan oleh satu *usecase* atau lebih. Aktivitas menggambarkan proses yang berjalan, sementara *usecase* menggambarkan bagaimana aktor menggunakan sistem untuk melakukan aktivitas. Standar UML menggunakan segiempat dengan sudut membulat untuk menggambarkan aktivitas. *Decision* digunakan untuk menggambarkan *behaviour* pada kondisi tertentu. Untuk mengilustrasikan proses-proses paralel (*fork* dan *join*) digunakan titik sinkronisasi yang dapat berupa titik, garis horizontal atau vertikal. *Activity* diagram dapat dibagi menjadi beberapa *object swimlane* untuk menggambarkan objek mana yang bertanggung jawab untuk aktivitas tertentu [15].

2.11 Pengujian Unit

Pengujian unit merupakan pengujian yang menguji bagian terkecil dari kode, biasanya fungsi individu, sendirian, dan terisolasi. Ketika pengujian menggunakan beberapa sumber daya eksternal, seperti database, hal ini bukanlah pengujian unit.

Pengujian unit pada dasarnya hanya memberikan fungsi yang diuji dengan beberapa input, dan kemudian memeriksa apakah output dari fungsi benar [17].