BAB II

TINJAUAN PUSTAKA DAN DASAR TEORI

2.1. Tinjauan Pustaka

Dalam penyusunan skripsi ini, penulis mengambil referensi dari beberapa buku untuk menggali informasi tentang teori yang berkaitan dengan judul yang digunakan sebagai landasan teori. Selain itu penulis juga mengambil referensi dari peneliti sebelumnya mengenai implementasi arsitektur *microservices* dan *graphQL*. Adapun peneliti - penelitian sebelumnya yang menjadi tinjauan pustaka dalam penulisan ini adalah sebagai berikut :

Tabel 1. Tinjauan Pustaka

No	Peneliti	Objek	Metode	Hasil
1	Satyabudhi, Riza Ega (2019)	Go-Life	Microservices dan GraphQL	Dari penelitian ini, ditemukan bahwa pada arsitektur *microservice* Go-Life, setelah mengimplementasikan *GraphQL*, waktu respons yang diperlukan untuk menangani permintaan data relasional dan non-relasional dari klien dapat diturunkan, karena masalah N+1 telah diselesaikan.

2	Junilla, Adissa Vintha (2021)	Sistem Informasi Akademik UIN Jakarta (Studi Kasus : AIS Mahasiswa UIN Jakarta)	Perancangan UI/UX *Microservices* dan Metode Perancangan *Five Planes*	Menghasilkan *prototype* pengembangan UI/UX yang tepat sesuai kegunaan AIS UIN Jakarta sebagai interaktif mahasiswa. *Prototype* yang telah dibangun dinamakan AIS For Student Pada hasil evaluasi dengan menggunakan metode UEQ, AIS *For Student* pada skala Daya Tarik, Kejelasan, Ketepatan, stimulasi dan kebaruan mendapat hasil dengan kategori *Good* (Baik).
3	Irawan, Aji Karuniadi (2020)	Input Nilai Praktikum Mahasiswa STMIK Akakom Yogyakarta	*Microservices* dan *RESTFULL API*	Penelitian ini menghasilkan sebuah sistem yang lebih *flexible* baik dalam pengembangan atau *maintenance* karena menggunakan teknologi *Microservice* yang memisahkan antara bagian *Frontend* dengan bagian *Backend* sehingga performa aplikasi menjadi lebih baik, cepat dan *scale*.

4	Qomaruddin, Ahmad (2018)	Portal Akademik PP AL-Munawwir	*Microservices* dan *RESTFULL API*	Hasil dari penelitian ini adalah aplikasi portal akademik Pondok Pesantren Al-Munawwir dengan implementasi arsitektur *Microservice* menggunakan RESTful API. Aplikasi ini memiliki beberapa layanan seperti jadwal kegiatan santri, jadwal kajian, nilai dan jurnal-jurnal.
5	Sastra, Arief Permana, (2020)	Analisis Perbandingan Metode *GraphQL* Dan Metode *REST API* Pada Teknologi *Nodejs*	*GraphQL*, *REST API* dan *Nodejs*	Penelitan ini menyimpulkan bahwa berdasarkan parameter-parameter yang digunakan metode graphql lebih unggul dibandingkan metode *rest api* akan tetapi dapat disarankan oleh penulis bahwa untuk web aplikasi yang kompleks dapat menggunakan metode *rest api* karena dapat diandalkan design urlnya.

2.2. Dasar Teori

Untuk mendukung penelitian ini, maka perlu dikemukakan hal - hal atau teori-teori yang berkaitan dengan permasalahan dan ruang lingkup pembahasan sebagai landasan dalam penelitian.

2.2.1. Framework

Framework secara sederhana dapat didefinisikan sebagai gabungan dari fungsi atau langkah dan kelas untuk mencapai suatu tujuan yang sudah siap untuk digunakan sehingga memberikan kemudahan dan efisiensi waktu kepada *programmer* dalam membuat program tanpa harus membangun fungsi atau *class* dari awal.

Framework berbeda dengan *Content Management System* (CMS), karena CMS cukup diinstall dan dijalankan saja, sedangkan *framework* tidak demikian. Menggunakan *framework* untuk membuat program, pengguna atau *programmer* diharuskan melakukan *coding* atau penulisan kode-kode program pada lingkungan *framework* tersebut. Hal penting yang perlu disadari adalah *programmer* harus mengerti dan memahami terlebih dahulu tentang alur *framework* yang akan digunakan.

Sebagian besar framework yang telah ada mengimplementasikan pola desain atau *Model-View-Controller* (MVC), yang memisahkan bagian kode untuk penanganan proses bisnis dengan bagian kode untuk keperluan presentasi (tampilan). Menurut Hofmeister (1999), pola MVC terbukti efektif untuk generasi modul. Adapun komponen MVC adalah sebagai berikut:

a) *Model*, merupakan bagian yang menangani hal yang berhubungan dengan pengolahan dan manipulasi data, seperti menambah, merubah, mengambil, dan menghapus data yang ada pada basis data.
b) *View*, merupakan bagian yang mengatur tampilan sistem informasi yang digunakan untuk berinteraksi dengan pengguna.
c) *Controller*, merupakan bagian yang menghubungkan Model dan View secara langsung

2.2.2. CodeIgniter

Codeigniter adalah sebuah *framework* PHP yang dikatakan mempunyai waktu eksekusi lebih cepat bila dibandingkan dengan *framework* PHP yang lain. Dalam *Codeigniter* tidak terdapat lisensi dan menggunakan konsep MVC (*Model View Controller*), dimana konsep tersebut merupakan pola konsep modern *framework* yang banyak diaplikasikan saat ini. *Codeigniter* pertama kali dikembangkan pada tahun 2006 oleh Rick Ellis pendiri *EllisLab.com*.

Dalam mengembangkan *framework codeigniter* Rick Ellis memiliki tujuan yaitu untuk membentuk sebuah struktur yang dapat dipergunakan dalam mengembangkan sebuah *website* agar dapat terselesaikan dengan lebih cepat. Langkah yang dilakukan yaitu dengan membangun berbagai *library* yang diperlukan dalam pengembangan *website* sesuai dengan kebutuhan.

Kelebihan menggunakan *framework codeigniter* dalam mengembangkan sebuah *website*, di antaranya yaitu:

a) Bebas digunakan tanpa lisensi
b) Dapat dijalankan di berbagai platform

c) Penggunaan waktu yang lebih efisien

d) Menggunakan konsep MVC

e) Mudah digunakan.

f) Framework yang lengkap

g) Mudah dipelajari karena terdapat dokumentasi mengenai cara penggunaan codeigniter

h) Dapat dijalankan pada PHP versi 4 ke atas

2.2.3. MySQL

SQL (*Structured Query Language*) adalah bahasa standar yang digunakan untuk mengakses *server database*. Semenjak tahun 70-an, bahasa ini telah dikembangkan oleh IBM, yang kemudian diikuti dengan adanya *Oracle*, *Informix*, dan *Sybase*. Dengan SQL, proses akses *database* menjadi lebih *user friendly* dibandingkan dengan misalnya *dBase* ataupun *Clipper* yang masih menggunakan perintah-perintah pemrograman murni.

MySQL adalah sebuah *server database* SQL *multiuser* dan *multi-threaded*. SQL sendiri adalah salah satu bahasa pemrograman *database* yang paling populer di dunia. Implementasi program *server database* ini adalah program daemon ‘mysqld’ dan beberapa program lain serta beberapa pustaka.

MySQL adalah *database server* yang sangat ideal untuk data segala ukuran. Dengan kemampuannya yang dapat bekerja di lingkungan Unix maupun Win32 dan sifat yang *Open Source Freeware* (di bawah kungkungan GNU, *General Public License*), *MySQL* menjadi pilihan yang tepat bagi pengembang aplikasi kelas menengah ke bawah dan kelas korporat. Kemampuan paling menonjol *MySQL*

server adalah dalam hal kecepatannya yang sangat tinggi dalam melakukan proses data, *multithreaded*, *multi-user*, dan sangat mudah dalam melakukan *query* dibandingkan *SQL server* yang lain.

Beberapa keuntungan dalam menggunakan *database MySQL server* antara lain:

a) MySQL merupakan program yang *multi-threaded*, sehingga dapat dipasang pada *server* yang memiliki *multi-CPU*.
b) Didukung program-program umum seperti C, C++, Java, Perl, PHP, Python, TCL APIs, dan lain-lain.
c) Bekerja pada berbagai *platform*. (Tersedia berbagai versi untuk berbagai sistem operasi).
d) Memiliki jenis kolom yang cukup banyak sehingga memudahkan konfigurasi sistem basis data.
e) Memiliki sistem keamanan yang cukup baik dengan verifikasi host.
f) Mendukung ODBC untuk sistem operasi *Microsoft Windows*.
g) Mendukung record yang memiliki kolom dengan panjang tetap atau panjang bervariasi, dan masih banyak keunggulan lainnya.
h) *MySQL* merupakan software yang gratis untuk digunakan.
i) *MySQL* dan PHP saling. Maksudnya adalah pembuatan *database* menggunakan sintak PHP dapat dibuat. Sedangkan input yang dimasukkan melalui aplikasi web yang menggunakan *script server-side* seperti PHP dapat langsung dimasukkan ke database *MySQL* yang ada di server dan tentunya web tersebut berada di sebuah *web server*.

2.2.4. UML

UML adalah visualisasi dari pemodelan dan komunikasi dari sebuah sistem dalam bentuk diagram dan kode-kode pendukung. Fungsi dari UML hanya terbatas pada pemodelan, sehingga penggunaan UML tidak fokus pada metode tertentu meskipun pada umumnya UML digunakan hanya pada metode berorientasi obyek.

- *Use Case Diagram*

Shalahuddin (2014: 155) mengemukakan bahwa diagram *use case* adalah pemodelan untuk tingkah laku (behavior) sistem informasi ingin dikembangkan. Interaksi antara aktor dengan sistem informasi yang akan dikembankan dideskripsikan oleh diagram *use case*. Sedangkan menurut Hamilton (2006: 20), diagram *use case* merupakan diagram yang memodelkan interaksi antara sistem dengan pengguna. Kriteria penamaan pada diagram *use case* yaitu nama dituliskan sesederhana mungkin serta mudah untuk dipahami.

- *Activity Diagram*

Activity diagram adalah gambaran dari *workflow* (aliran kerja) atau aktivitas dari sebuah sistem. Diagram aktivitas menggambarkan aktivitas yang dapat dilakukan oleh sistem, bukan apa yang dilakukan oleh aktor. Shalahuddin (2014: 161). Sedangkan menurut Fowler (2004: 163), *Activity diagram* merupakan suatu diagram yang berisi gambaran mengenai model alur kerja, prosedur dan skenario dari sebuah sistem.

- *Class Diagram*

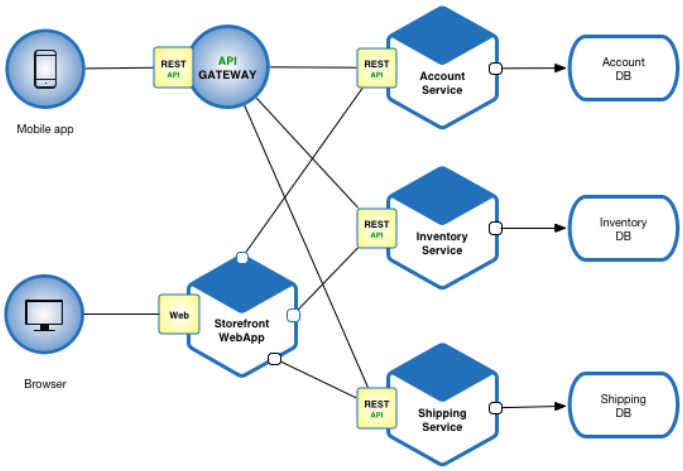
Class diagram merupakan diagram yang berisi gambaran dari struktur kelaskelas yang akan diimplementasikan pada sistem yang akan dibuat.

(Shalahuddin, 2014: 141). Di dalam kelas terdapat dua komponen yaitu atribut dan operasi atau metode. Atribut adalah variabel yang dimiliki oleh suatu kelas sedangkan operasi atau metode adalah fungsi yang dimiliki oleh kelas tersebut.

2.2.5. Microservices

Microservices adalah komponen kecil perangkat lunak yang khusus dalam satu tugas tertentu dan bekerja sama untuk mencapai tugas tingkat tinggi (Gonzalez, 2016). Aplikasi diatur sedemikian rupa sehingga saling terpisah menjadi *service-service* kecil yang *independent*, berfungsi spesifik (*high cohesion*) dan tidak saling bergantung pada komponen program lainnya (*loose coupling*), dengan antarmuka API (*Application Programming Interface*) (Newman,2015). Secara sederhana arsitektur aplikasi *Microservices* menggunakan desain yang memecah aplikasi berdasarkan fungsinya secara spesifik. Tidak sekedar memisahkan berdasarkan *user-role* atau *subdomain* saja, tetapi aplikasi akan di *breakdown* lebih rinci dari segi fungsionalitasnya. Aplikasi dirancang agar setiap fungsi bekerja secara *independent*. Dan setiap fungsi dapat menggunakan teknologi stack sesuai dengan kebutuhan yang berarti akan ada teknologi yang berbeda dalam satu aplikasi besar (Triwahyudhi, 2016). *Microservices* mendapat banyak perhatian di akhir ini. *Microservices* itu sendiri muncul karena didorong oleh beberapa faktor diantaranya untuk mengatasi keterbatasan gaya SOA (Pahl & Jamshidi, 2016). Hal ini merupakan langkah selanjutnya dalam evolusi aplikasi perusahaan. *Microservices* muncul dari *domain driven design*, *continuous delivery*, *virtualisasi* berdasarkan permintaan, infrastruktur otomatisasi, dan kebutuhan akan tim kecil yang

mengelola keseluruhan dari siklus produk (Livora, 2016). Adapun aplikasi terdiri dari beberapa layanan seperti 2.1 di bawah ini :



Gambar 2.1 *Microservices*

Dari gambar 2.1 diatas terlihat bahwa konsep dari *microservices* yaitu *service-service* kecil yang *independent*, *service* dapat berjalan sendiri dan dapat mengelola sendiri *service* yang ada tanpa berkaitan dengan *service* lain. Dalam *microservices* menurut penulis Mike Amundsen, Irakli Nadareishvili, Ronnie Mitra, dan Matt McLarty (Slocum, 2018) mengemukakan sifat aplikasi *microservices* adalah sebagai berikut :

1. Ukurannya kecil
2. Dibatasi oleh konteks
3. Berkembang secara otonomi
4. Dapat di deploy secara independen
5. Desentralisasi
6. Dibangun dan dirilis dengan proses otomatis

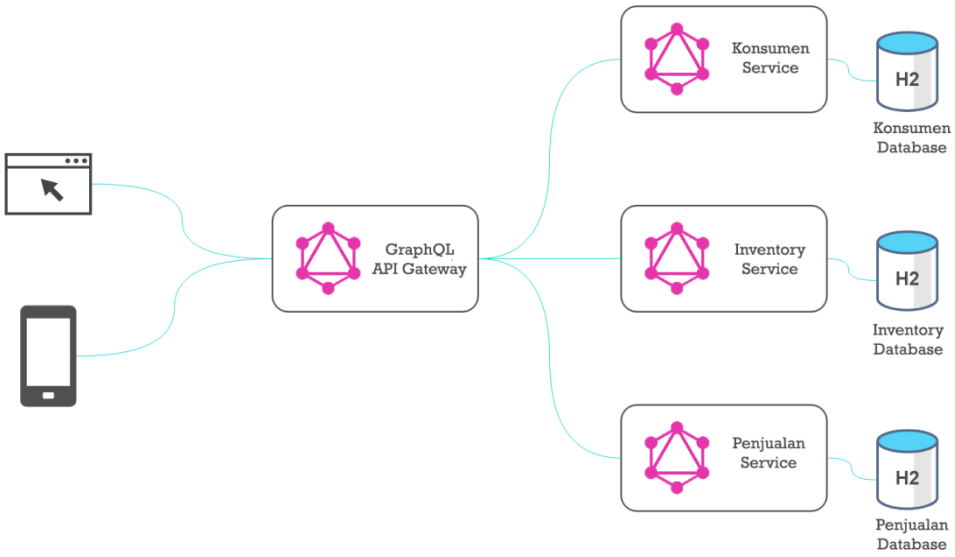
2.2.6. API Gateway

API Gateway adalah jenis layanan dalam arsitektur *microservices* yang menyediakan lapisan bersama dan API bagi *client* untuk berkomunikasi antar *service*. API dapat mengarahkan permintaan, mentransformasikan protokol, menggabungkan data dan menerapkan logika bersama seperti otentikasi. *API Gateway* berfungsi sebagai gerbang utama masuk ke layanan *microservices*. *Microservices* merupakan layanan dimana dapat menerapkan berbagai macam teknologi, sehingga tim yang membangun *microservices* dapat menggunakan berbagai bahasa pemrograman, *database* dan protokol berbeda untuk menerapkan *microservices*. Dari situlah disini peran *API Gateway* untuk menyediakan lapisan bersama untuk menangani perbedaan protokol layanan (Marton, 2017).

2.2.7. GraphQL

GraphQL merupakan sebuah bahasa kueri untuk API yang dikembangkan oleh Facebook dan dipergunakan dalam komunikasi antara klien dan server, dimana kueri hanya meminta data yang dibutuhkan oleh klien, sehingga klien dapat menerima seluruh data yang dibutuhkan dari server dalam satu kali request. *GraphQL* menyediakan *server side runtime* untuk mengeksekusi *type queries*. Setiap layanan *GraphQL* mendefinisikan tipenya pada *GraphQL Schema*. *GraphQL* juga memberi deskripsi data yang lengkap dan dapat dipahami oleh API, memungkinkan klien untuk meminta apa yang mereka butuhkan dan tidak lebih, sehingga dapat menghindari *overfetching* dan *undefetching* . Selain dapat menerima *JSON response*, *GraphQL* juga hanya memerlukan sebuah endpoint tunggal, yang lebih praktis dibandingkan dengan REST. *GraphQL* juga memiliki pustaka server

dalam berbagai bahasa yang berbeda termasuk *C#*, *Clojure*, *Elixir*, *Erlang*, *GO*, *Groovy*, *Java*, *JavaScript*, *.Net*, *PHP*, *Python*, *Scala*, dan *Ruby* (Porcello & Banks, 2018).



Gambar 2.2 *GraphQL*

Dalam (*The GraphQL Foundation*, 2020), Beberapa keunggulan yang dimiliki *GraphQL*, yaitu :

- Kueri dari *GraphQL* selalu mengembalikan hasil yang dapat diprediksi, membuat aplikasi yang mengimplementasikan *GraphQL* berjalan cepat dan stabil dikarenakan *GraphQL* mengontrol data alih-alih server.
- *GraphQL* API mendapatkan semua data yang dibutuhkan aplikasi dalam satu permintaan (*single request*). Dengan menggunakan *GraphQL*, aplikasi dapat berjalan dengan cepat bahkan pada koneksi jaringan yang lambat.
- *GraphQL* API diatur berdasarkan *types* dan *fields*, bukan dengan *endpoints*. *GraphQL* menggunakan *types* untuk memastikan aplikasi hanya meminta data yang diperlukan dan menghindari kesalahan yang dapat terjadi. Selain

itu, aplikasi dapat menggunakan *types* untuk menghindari penulisan manual *parsing code*.

- *GraphQL* menyediakan beberapa tools yang dapat memudahkan interaksi dengan *GraphQL* API, seperti *GraphQL* dan *GraphQL Playground. Tools* tersebut memudahkan pengguna GraphQL dalam mengetahui data apa saja yang dapat di-request dari API tanpa meninggalkan editor, membantu dalam menemukan masalah-masalah potensial sebelum mengirimkan kueri, dan bertanggung jawab dalam meningkatkan kecerdasan kode.
- *GraphQL* memungkinkan pengembang aplikasi untuk mengembangkan API tanpa memperbaharui API version, dimana pengembang dapat menambahkan *fields* dan *types* baru ke *GraphQL* API tanpa memengaruhi kueri yang sudah ada, sehingga memberikan aplikasi akses yang berkelanjutan ke fitur-fitur baru dan membuat kode server yang lebih bersih dan dapat dengan mudah dikelola.
- *GraphQL* membuat seluruh API seragam tanpa dibatasi oleh mesin penyimpanan tertentu. Pengembang dapat menyediakan fungsi untuk setiap fields pada type system, kemudian *GraphQL* memanggil fungsi-fungsi tersebut dengan konkurensi yang optimal.

2.2.8. GraphQL Schema

Server *GraphQL* menggunakan schema untuk menggambarkan bentuk dari grafik data yang dibuat. *Schema* mendefinisikan *hierarki type* dan *field* yang diisi dari penyimpanan data *back-end*. Schema juga menentukan secara spesifik kueri

dan mutasi apa yang tersedia bagi klien untuk dieksekusi terhadap grafik data (Stemmler, 2020b).

Untuk memfasilitasi dalam mendefinisikan type, *GraphQL* dilengkapi dengan bahasa yang dapat digunakan untuk mendefinisikan schema, yaang disebut *Schema Definition Language*, atau SDL. *GraphQL* SDL juga tidak memperdulikan apa bahasa atau *framework* yang digunakan untuk membangun aplikasi. *GraphQL Schema Document* adalah dokumen teks yang menentukan jenis yang tersedia dalam suatu aplikasi, dan digunakan oleh klien dan server untuk memvalidasi permintaan *GraphQL* (Porcello & Banks, 2018).

GraphQL SDL memiliki berbagai macam komponen di dalamnya, yaitu sebagai berikut (Porcello & Banks, 2018):

1. *Types*

Type merupakan unit utama dari *GraphQL Schema*. *Type* merepresentasikan data dari aplikasi. Sebuah type memiliki fields yang merepresentasikan data yang berhubungan dengan setiap objek, yang mana setiap field mengembalikan spesifik tipe data (dapat berarti integer atau string, namun dapat juga berupa tipe data buatan atau list tipe). Gambar 2.2 menunjukkan bagaimana bentuk type pada *GraphQL*.

```
type Photo {
  id: ID!
  name: String!
  url: String!
  description: String
}
```

Gambar 3.3 Contoh *type* pada *schema*

2. *Scalar Types*

Scalar types (seperti Int, Float, String, Boolean, ID) merupakan bagian dari *GraphQL* yang sangat bermanfaat, Namun ada kalanya diperlukan custom scalar type untuk keperluan tertentu. *Scalar type* bukanlah tipe objek, dikarenakan scalar type tidak memiliki fields. Contoh dari penggunaan custom scalar type seperti pada Gambar 2.3.

```
scalar DateTime

type Photo {
  id: ID!
  name: String!
  url: String!
  description: String
  created: DateTime!
}
```

Gambar 2.4 Penggunaan *custom scalar type DateTime*

3. *Enums*

Enumeration atau enums adalah *scalar type* yang memungkinkan field untuk mengembalikan set nilai string yang terbatas. Contoh dari enum dapat dilihat pada Gambar 2.4.

```
enum PhotoCategory {
  SELFIE
  PORTRAIT
  ACTION
  LANDSCAPE
  GRAPHIC
}
```

Gambar 2.5 Contoh *enum* dengan nama *PhotoCategory*

Pada Gambar 2.4, enum yang dinamakan PhotoCategory memiliki lima set nilai string, yaitu : SELFIE, PORTRAIT, ACTION, LANDSCAPE, dan GRAPHIC. PhotoCategory akan digunakan sebagai tipe data dari field ‘category’, yang akan mengembalikan satu dari lima nilai yang sudah ditetapkan.

```
type Photo {
  id: ID!
  name: String!
  url: String!
  description: String
  created: DateTime!
  category: PhotoCategory!
}
```

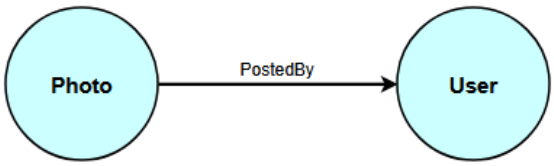
Gambar 2.6 Penggunaan *enum PhotoCategory* pada *field category*

4. Connections and Lists

Kemampuan untuk menghubungkan data dan permintaan beberapa jenis data terkait adalah fitur yang sangat penting. Saat membuat suatu daftar tipe objek khusus, *GraphQL* menyediakan fitur canggih untuk membantu dalam menghubungkan objek satu sama lain, yaitu sebagai berikut :

a. One-to-One Connections

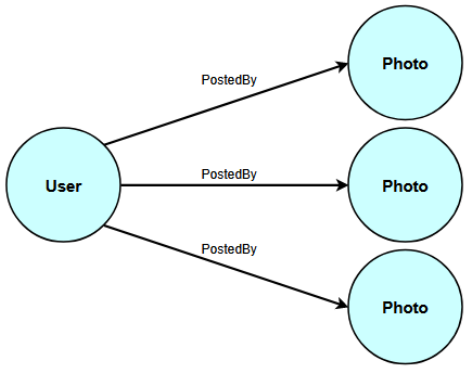
One-to-one connections menghubungkan *single object type* dengan *single object type* lainnya.



Gambar 2.7 *One-to-One Connections*

b. One-to-Many Connections

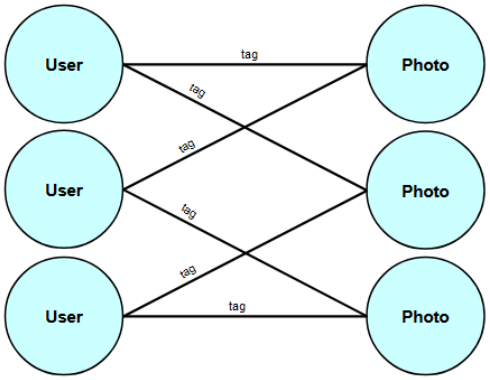
One-to-many connections menghubungkan *single object type* dengan banyak *object types*.



Gambar 2.8 *One-to-Many Connections*

c. *Many-to-Many Connections*

Many-to-many connections menghubungkan banyak *object types* dengan banyak *object types.*



Gambar 2.9 *Many-to-Many Connections*

5. *List of Different Types*

Dalam *GraphQL*, list tidak selalu mengembalikan type yang sama. Ada dua cara untuk menangani bidang yang dapat berisi banyak types dalam sebuah schema di *GraphQL*, yaitu dengan *unions* dan *interfaces*.

a. *Union types*, atau tipe gabungan adalah tipe yang dapat digunakan untuk mengembalikan salah satu dari beberapa tipe yang berbeda. Union memungkinkan untuk menggabungkan banyak tipe dalam satu kesatuan sesuai keinginan.

```
union AgendaItem = StudyGroup | Workout
type StudyGroup {
  name: String!
  subject: String
  students: [User!]!
}

type Workout {
  name: String!
  reps: Int!
}

type Query {
  agenda: [AgendaItem!]!
}
```

Gambar 2.10 Penggunaan *union* pada *schema*

b. *Interfaces*, merupakan tipe abstrak yang dapat diimplementasikan oleh tipe objek. Sebuah interface mendefinisikan seluruh fields yang harus disertakan dalam objek apa pun yang mengimplementasikannya. *Interface* adalah cara terbaik untuk menangani kode dalam schema, dikarenakan cara ini memastikan bahwa tipe-tipe tertentu selalu menyertakan fields spesifik yang dapat dikueri, tanpa peduli tipe yang dikembalikan.

Secara umum, *union types* sangat efektif digunakan jika objek berisi fields yang sama sekali berbeda, sedangkan *interface* lebih efektif digunakan jika tipe objek harus berisi field tertentu untuk dapat berinteraksi dengan tipe objek lain.

6. Arguments

Arguments merupakan pasangan *key-value* yang terkait dengan field kueri. *Argument* dapat ditambahkan ke dalam setiap field pada *GraphQL*, dan sama

seperti field, *Argument* juga memerlukan tipe data, baik dengan scalar type maupun *object types* yang tersedia dalam *schema*. Arguments memungkinkan pengiriman data yang dapat mempengaruhi hasil dari operasi *GraphQL*.

Dalam pengembalian data, *GraphQL* arguments dapat digunakan untuk mengontrol jumlah data yang perlu dikembalikan dari kueri. Proses ini disebut Data Paging, dikarenakan sejumlah halaman tertentu dikembalikan untuk mewakili satu halaman data.

Ketika melakukan kueri terhadap list data, *GraphQL* argument dapat digunakan untuk mengatur bagaimana list data yang dikembalikan dapat tersortir. Selain itu, argument juga dapat digunakan untuk filtering data juga.

7. Mutations

Mutations harus didefinisikan dalam schema. Seperti halnya kueri, mutations juga didefinisikan dengan *custom object* type-nya sendiri dan ditambahkan ke dalam schema. Perbedaan dalam mendefinisikan mutations dan kueri terletak pada tujuannya, dimana mutations perlu dibuat hanya apabila suatu action atau event dapat merubah sesuatu tentang keadaan aplikasi.

```
type Mutation {
  postPhoto(
    name: String!
    description: String
    category: PhotoCategory=PORTRAIT
  ): Photo!
}

schema {
  query: Query
  mutation: Mutation
}
```

Gambar 2.11 Penggunaan *mutation* dalam *schema*

Mutations harus terdiri dari segala sesuatu yang dapat dilakukan user dengan service milik mereka. Sehingga, dapat diartikan bahwa mutations merupakan semua actions yang dapat digunakan oleh user pada aplikasi mereka.

8. Input Types

Input type merupakan tipe data yang mirip seperti *GraphQL object type*, namun hanya digunakan untuk input *arguments*. Input types merupakan kunci untuk mengatur dan menulis *GraphQL* schema yang jelas. Selain dapat digunakan sebagai argument pada field manapun, Input type juga dapat digunakan untuk meningkatkan data paging dan data filtering pada aplikasi. Input type meningkatkan dokumentasi schema yang dihasilkan *GraphQL* dan *GraphQL Playground* secara otomatis, sehingga membuat API yang digunakan lebih mudah dipahami dan dipelajari, serta akan membuat klien memiliki kekuatan untuk menjalankan kueri yang terorganisir.

```
input Photofilter {
  category: PhotoCategory
  createdBetween: DateRange
  tanggedUsers: [ID!]
  searchText: String
}
input DateRange {
  start: DateTime!
  end: DateTime!
}
input DataPage {
  first: Int = 25
  start: Int = 0
}
input DataSort {
  sort: SortDirection = DESCENDING
  sortBy: SortablePhotoField = created
}
type User {
  postedPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
  inPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
}
type Photo {
  taggedUsers(sorting:DataSort): [User!]!
}
type QUery {
  allUsers(paging:DataPage sorting:DataSort): [User!]!
  allPhotos(filter:PhotoFilter paging:DataPage sorting:DataSort): [Photo!]!
}
```

Gambar 2.12 Penggunaan *input type* pada *schema*

9. Return Types

GraphQL schema juga dilengkapi dengan fitur *custom return types*, yang mempunyai fungsi seperti untuk mengetahui berapa lama waktu yang dibutuhkan sebuah kueri untuk mengirim respon, atau berapa banyak hasil yang ditemukan dalam respon tertentu selain dari query data *payload*.

10. Subscriptions

Subscriptions sama seperti *object type* lainnya dalam *GraphQL* SDL. Sama seperti kueri dan mutations, *subscriptions* juga memberi keuntungan untuk arguments. *Subscriptions* merupakan solusi yang tepat ketika menangani data *real time* menjadi hal penting.

11. Schema Documentation

GraphQL schema memiliki fitur untuk menambahkan deskripsi opsional pada setiap field yang mana akan memberikan informasi tambahan mengenai tipe schema dan fields, sehingga memudahkan dalam kolaborasi tim.

```
"""
The user's first and last name
"""
name: String
"""
A url for the user's GitHub profile photo
"""
avatar: String
"""
All of the photos posted by this user
"""
postedPhotos: [Photo!]!
"""
All of the photos in which this user appears
"""
inPhotos: [Photo!]!
```

Gambar 2.13 *Three quotation marks* untuk menambahkan komentar

Dengan memberi three quotation marks di antara setiap komentar pada type dan field sama halnya dengan memberikan kamus untuk user dalam menggunakan API. Seluruh catatan dokumentasi ini tercantum di dalam *schema documentation* pada *GraphQL Playground* atau *GraphQL*.

2.2.9. Schema Stitching

Schema stitching merupakan proses pembuatan sebuah *GraphQL* schema dari beberapa *GraphQL* API yang mendasarinya. *Schema stitching* digunakan untuk menggabungkan beberapa *GraphQL schema* bersama dan menghasilkan sebuah *schema* baru yang mengetahui cara mendelegasikan bagian kueri ke subschemas yang relevan. *Use case* dari schema stitching sendiri adalah untuk membuat *codebase* besar dapat diatur dengan memisahkan API yang besar menjadi

layanan *GraphQL* yang lebih kecil (The Guild, 2020). Schema stitching juga dapat digunakan untuk menggabungkan *local GraphQL schemas* dan *remote GraphQL schemas* serta memperluas tipe dari schema yang tergabung dengan field baru, atau menambahkan fields ataupun *entities* dari *schema* lainnya (Touronen, 2019).

Schema stitching diimplementasikan di *Apollo GraphQL library*. *Apollo library* menyediakan fungsi ToSchema yang dapat digunakan untuk menambahkan entitas *schema* lain ke dalam entitas *extended schema*. Fungsi ini sangat bermanfaat seperti untuk mengisi entitas field ID dengan entitas aktual dari *schema* yang lain (Touronen, 2019).