BAB 2

TINJAUAN PUSTAKA DAN DASAR TEORI

2.1 Tinjauan Pustaka

Penelitian ini meggunakan beberapa sumber pustaka yang berhubungan dengan kasus atau metode yang akan diteliiti. Diantaranya yaitu:

Heri Purnama(2016), telah melakukan penelitian tentang Aplikasi Pengolahan Skripsi Di STMIK AKAKOM YOGYAKARTA Menggunakan Arsitektur *Miroservice* Dengan *Node.js*. Aplikasi dibuat bertujuan untuk mengecek duplikasi skripsi dengan berdasarkan judul dari skripsi. Dibuat dengan bahasas pemograman *JavaScript* pada *Node.js*, menggunakan database *MongoDB* yang mengimplementasikan arsitektur *Microservice*.

Ahmad Anwar (2017), telah melakukan penelitian tentang Implementasi Basis Data NoSQL *Cassandra* Pada Sistem Perngarsipan Dokumen Skripsi Dan Tugs Akhir Di Perpustakaan STMIK AKAKOM Berbasis Web Dalam *Private Cloud.*

Pada penelitian yang akan penulis lakukan dengan judul "Implementasi Arsitektur Microservice Menggunakan RESTful API Untuk Portal Akademik PP.Al-Munawwir", penelitian ini bertujuan untuk mengimplementasikan arsitektur *microservice* untuk membangun aplikasi web dengan studi kasus portal akademik PP Almunawwir.

Tabel tinjauan pustaka merupakan tabel yang dibuat untuk mendefinisikan penelitian yang sebelumnya hampir sama dilakukan dengan penelitian yang diajukan saat ini, adapun perbandingan yang menjadi tabel tinjauan pustaka penelitian yakni dijabarkan pada Tabel 2.1:

Tabel 2.1 Perbandingan Penelitian

Penulis	Topik Penelitian	Bahasa Pemrograman	Teknologi	Database
Heri Purnama (2016)	Pengolahan Skirpsi Menggunakan Arsitektur Microservice Dengan Node.js	Javascript	Microservice, Node.js, Private Cloud (Docker)	NoSQL-MongoDB
Ahmad Anwar (2017)	Arsip Dokumen Skripsi dan Tugas Akhir Dengan Basis Data NoSQL Cassandra	Python	Private Cloud (Docker)	NoSQL-Cassandra Apache
Ahmad Qomaruddin (2018)	Implementasi Arsitektur Microservice Menggunakan RESTful API Pada Portal Akademik	Go	Microservice, RESTful API	Cockroach DB

2.2 Dasar Teori

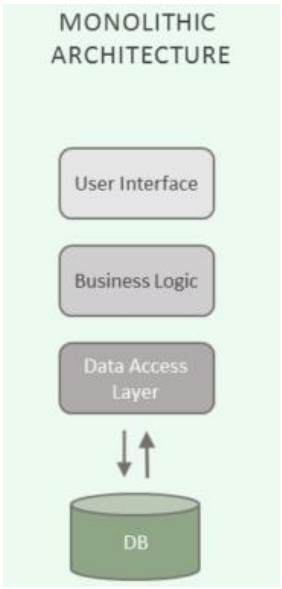
2.2.1 Monolithic Architecture

Aristektur *monolitik* adalah model pemrograman tradisional, yang berarti bahwa elemen program perangkat lunak terjalin dan saling bergantung satu sama lain.*(Margaret Rouse, 2016)*

Arsitektur *monolitik* ini menggunkan kode sumber dan teknologi yang serupa untuk menjalankan semua tugas-tugasnya.*(Boyke Dian Triwahyudi, 2016)*

Enterprise Application, dibangun dalam tiga bagian: database(terdiri dari banyak tabel biasanya dalam sistem manajemen basis data relasional), *user interface* pada sisi klien (terdiri dari halaman HTML yang berjalan di browser), dan sisi server aplikasi.

Aplikasi sisi server ini akan menangani permintaan *HTTP*, menjalankan beberapa logika spesifik domain, mengambil dan memperbarui data dari database, dan mengisi tampilan HTML untuk dikirim ke browser. Ini adalah arsitektur *monolitik*, sebuah *single logical executable*. Untuk membuat perubahan pada sistem, pengembang harus membuat dan menerapkan versi terbaru dari sisi server.



Gambar 2.1 *Monolithic Architecture*

2.2.2 Microservices Architecture

Microservices adalah sebuah pendekatan untuk mengembangkan aplikasi dengan rangkaian service-service yang kecil, yang mana setiap service berjalan pada prosesnya sendiri-sendiri. Setiap service dapat berkomunikasi dengan mekanisme yang ringan.

Tiap-tiap service yang dibuat harus mengenskapsulasi data dengan logika bisnis yang beroperasi pada data itu sendiri, dan hanya dapat diakses melalui *published service interface*. Tidak ada database yang dapat diakases secara langsung dari luar service dan tidak ada data yang *disharing* antara setiap service.

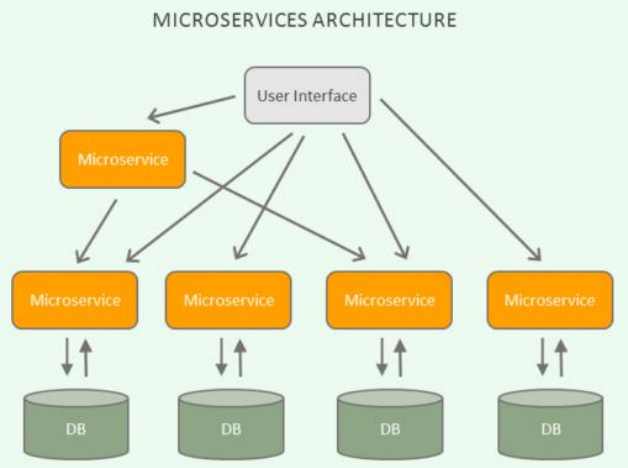
Dengan begitu, setiap service yang dibangun harus memiliki landasan domainnya sendiri-sendiri dan process *sharing* data domain yang satu dengan yang lainnya hanya dapat dilakukan melalui *published service interface.*

Microservices memperkuat struktur modular yang sangat penting bagi tim yang sangat besar. Menurut *Martin Fowler* ini adalah *key benefit* yang juga aneh jika dikatakan kelebihan, karena tidak ada alasan apapun mengapa *microservices* memiliki struktur modular yang lebih kuat daripada *monolithic*.

Dalam arsitektur *monolithic* pada umumnya, sangatlah mudah bagi *developer* untuk melewati batas. Umumnya digunakan untuk mencari jalan pintas dalam mengimplementasikan fitur dengan lebih cepat. Akan tetapi berakhir pada merusak struktur modular yang berimplikasi pada penurunan produktifitas Tim.

Service yang sederhana lebih mudah *dideploy* dan digunakan. Karena mereka berdiri sendiri, kecil kemungkinan kegagalan sistem terjadi saat salah satu service mengalami kesalahan.

Dengan *mikroservices*, kita dapat mencampur dan menggunakan beragam bahasa pemrograman, *framework*, dan teknologi penyimpanan database yang digunakan.*(Refactory, 2017)*



Gambar 2.2 *Microservices Architecture*

2.2.3 Go

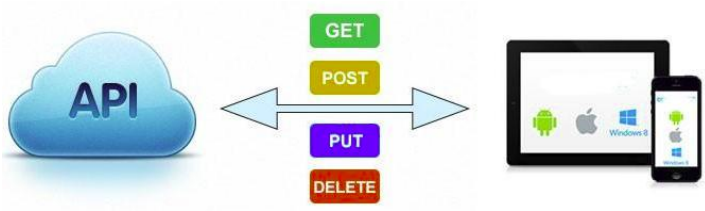
Golang atau yang sering disebut *Go*, adalah bahasa pemograman *open sourch* yang memudahkan dalam membangun software yang sederhana, andal dan efesien. *Go* adalah proyek *open source* untuk membuat program lebih produktif.*(Golang.org)*

Go mulai dikembangkan pada 2007 di *Google*, dan diperkenalkan ke publik pada tahun 2009. Tiga pengembang utama *Go* di *Google* adalah *Robert Griesemer, Rob Pike,* dan *Ken Thompson*. Tujuan mereka adalah membuat sebuah bahasa, berdasarkan sintaks bahasa pemrograman *C* yang longgar, yang akan menghilangkan *"extraneous garbage"* bahasa seperti *C++*. Oleh karena itu, *Go* menjauhkan banyak fitur dari bahasa modern lainnya, seperti metode dan operator *overloading*, *aritmatika pointer* dan jenis warisan. *Go* bukan sebuah bahasa

free-from, konvensinya menentukan banyak detail pemformatan, termasuk bagaimana identasi dan spasi digunakan. Bahasa tersebut mensyaratkan bahwa tidak satupun variabel yang dideklarasikan atau *libraries* yang diimpor tidak digunakan, dan semua pernyataan return bersifat wajib.*(Computer Hope, 2017)*

2.2.4 RESTful API

Appliction Programming Interface(API) adalah sebuah teknologi untuk memfasilitasi pertukaran informasi atau data antara dua atau lebih aplikasi perangkat lunak. API adalah antarmuka virtual antara dua fungsi perangkat lunak yang saling bekerja sama. (Yusa Inderapermana, 2017)



Gambar 2.3 Restful API

REST *(Representation State Transfer)* adalah satu implementasi dari webservice yaitu sebuah standar yang digunakan untuk pertukaran data antar aplikasi atau sistem. REST adalah suatu arsitektur metode komunikasi yang menggunakan protokol HTTP untuk pertukaran data. Dimana tujuannya adalah untu menjadikan

sistem yang memiliki performa yang baik, cepat dan mudah untuk dikembangkan(*scale*) terutama dalam pertukaran dan komunikasi data.

RESTful API merupakan implementasi dari API. RESTful juga disebut sebagai protokol untuk melakukan REST. RESTful API memiliki 4 komponen penting didalamnya diantaranya adalah:

a. URL *Design*, RESTful API diakses menggunakan protokol HTTP. Penamaan dan struktur URL yang konsisten akan menghasilkan API yang baik dan mudah untuk dimengerti *developer*.

b. HTTP *Verbs*, setiap *request* yang dilakukan terdapat metode yang dipakai agar server mengerti apa yang sedang *direquest client*, diantaranya adalah *GET, POST, PUT, DELETE.*

c. HTTP *Response Code*, adalah kode standarisasi dalam menginformasikan hasil *request* kepada *client*. Secara umum dbagi 3 kelompok yaitu 2xx(*response code* bahwa *request* berhasi), 4xx(*response code* bahwa *request* mengalami kesalahan pada sisi *client*) dan 5xx(*response code* bahwa *request* mengalami kesalahan pada sisi *server*).

d. Format *Response*, setiap *request* yang dilakukan *client* akan menerima data *response* dari server, *response* tersebut biasanya berupa data XML atau JSON. (Sepry Haryandi, 2016)

Cara kerja RESTful APi adalah REST *client* akan mengakses data/*resource* ke REST server dimana masing-masing *resource* atau data tersebut dibedakan oleh sebauh global ID atau URIs *(Universal Resource Identifiers)*. Data yang diberikan oleh REST server tersebut dapat berupa format text, JSON atau XML.

Cara kerja REST API yiatu pertama harus ada sebuah REST server yang akan menyediakan *resource*/data. Sebuah REST *client* akan membuat HTTP *request* ke server melalui sebuah global ID atau URIs dan server akan merespon dengan mengirimkan balik sebah HTTP respon sesuai yang diminta *client*.

Komponen dari HTTP *reequest:*

- HTTP *method* sesuai dengan tugasnya masing-masing
- URI untuk mengetahui lokasi data di server
- HTTP *version*, seperti HTTP V1.1
- *Request Header*, berisi metadata seperti *Authorization*, tipe *client* dan lainnya
- *Request Body*, data yang diberikan *client* ke server seperti URI *params*

Komponen dari HTTP response:

- *Response Code*, status server terhadap request yang diminta seperti 200, 404, dan lainnya
- HTTP *version*

- *Response Header* yang berisi meta data seperti *content type*, *cache tag* dan lainnya
- *Response Body*, data/*resource* yang diberikan oleh server baik beruppa text, json ataupun xml

(ngoding.net/2017)

2.2.5 CockroachDB

CockroachDB adalah database SQL terdistribusi. Tujuan desain utama adaah *scalability*, *consistency* dan *survivability* yang kuat (oleh karena itu dinamakan *Cockroach*).

CockroachDB adalah database SQL dapat bertahan, terukur dan sangat konsisten, menawarkan konsistensi yang kuat, SQL terdistribusi, transaksi kecepatan tinggi dan juga untuk membangun aplikasi hebat. Karena bantuan teknologi database terdistribusi, CockroachDB tidak bergantung pada *developer* untuk melakukan *tradeoffs* antara *time-to-market* dan *scalable* insfrastruktur.

CockroachDB memungkinkan *developer* untuk membangun aplikasi terukur yang dapat bertahan dari gangguan data center tanpa kesulitan.

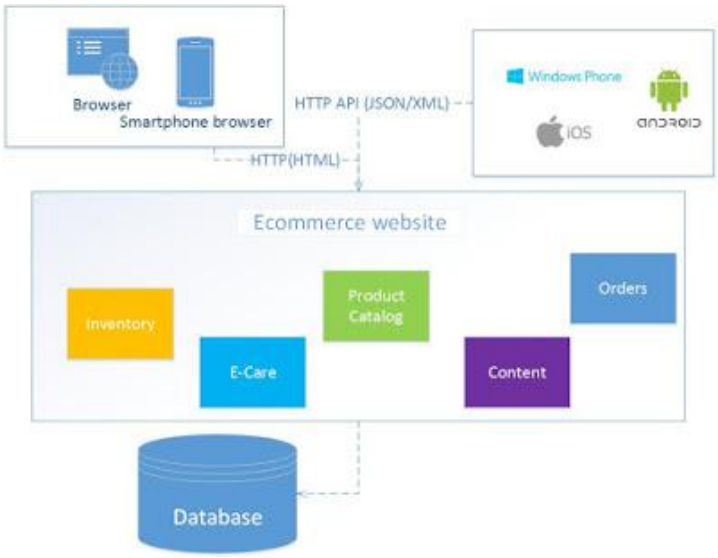
CockroachDB dilengkapi dengan salah satu sistem pemulihan *(disaster recovery)* terkuat di pasaran, yang berarti ketika data center atau infrastruktur *cloud* berjalan

offline tiba-tiba atau malfungsi bahkan sekecil apapun, aplikasi tetap menyala. *Obligasi* data center CockroachDB memberikan layanan otomatis dan transparan terus menerus dalam menghadapi pemadaman skala besar, aplikasi tidak memerlukan pengetahuan atau penanganan khusus *failover*. Fitur ini membuat software berjalan meski kondisinya tidak baik. CockroachDB adalah sebuah *scale-out*, SQL RDBMS mendukung transaksi ACID*(Atomicity, Consistency, Isolation and Durability)*. Ditambah fitur lainnya termasuk : pilihan untuk melihat riwayat data sekaligus merekamnya untuk penggunaan masa depan, proses *self-cleaning* untuk perangkat disk dan penyimpanan, sistem aplikasi *independent* dan *self-maintained* yang membangun diriya sendiri ketika data *mismanaged* atau dibiarkan tanpa ada *heads categoryzation* dan semacamnya, pilihan untuk mendapatkan perubahan Skema *high-definition* untuk *software*, navigasi *user interface* yang lancar, dan lain-lain. (Predictive Analytics Today, 2018)

2.2.6 Skema Arsitektur Monolithic Dan Microservices Pada Aplikasi Web

Skema Arsitektur *Monolithic* menggambarkan hubungan dalam sebuah aplikasi dalam satu wadah sedangkan skema Arsitektur *Microservices* menggambarkan hubungan antara satu layanan dengan layanan lain, yang digambarkan dalam implementasi suatu aplikasi web.

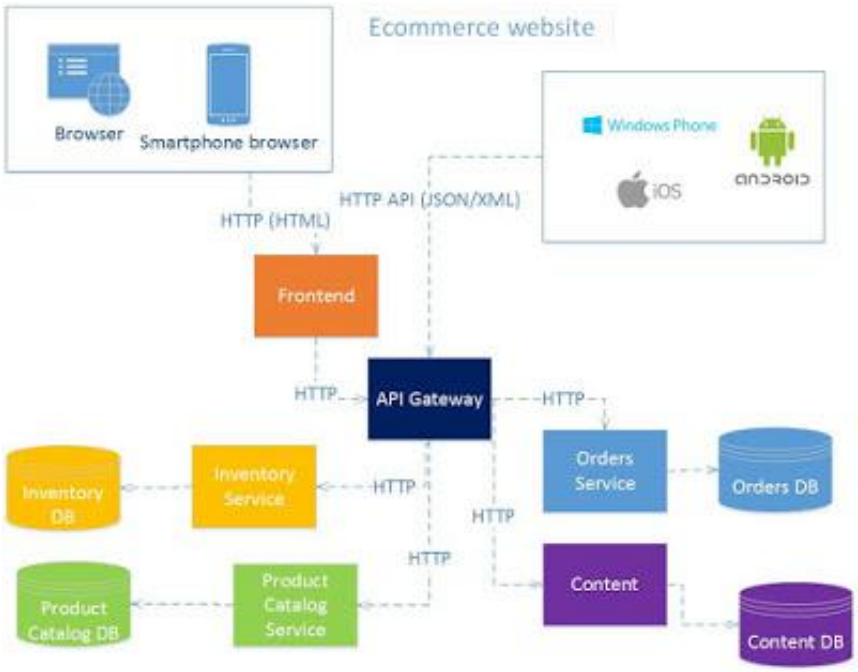
Robert Rusu dalam blognya pernah menggambarkan arsitektur perbedaan antara arsitektur *Microservice* dan arsitektur *Monolitik* pada sebuah aplikasi *e-commerce*. Secara umum semua aplikasi dari kategori ini memerlukan fungsionallitas untuk *browsing* melalui produk yang tersedia, membelinya dan melakukan pemesanan yang kemudian dikelola oleh administrator. Dan tentu saja sebagian besar situs *e-commerce* memiliki beberapa konten yang harus mudah diedit kapan saja dari *dasbor* administrasi.



Gambar 2.4 Skema Arsitektur Monolitik Pada Web E-commerce

Pada Gambar 2.4 disajikan sebuah aplikasi web *e-commerce* yang dibangun mengikuti arsitektur *monolithic*. Pada gambar diatas, data aplikasi web disimpan dalam database tunggal. Produk, konten situs web, pesanan yang ditempatkan oleh pelanggan dan informasi persediaan semua disimpan di tempat yang sama.

Scaling untuk jenis aplikasi ini bisa dilakukan baik secara horisontal maupun vertikal. Yang kemudian berarti bahwa mesin produksi akan memperbaiki perangkat keras dengan menambahkan memori RAM tambahan, penyimpanan disk atau menambahkan CPU yang lebih baik. *Scaling* vertikal dapat dicapai dengan menginstal versi yang sama dari aplikasi pada beberapa node dan menempatan *load balancer* antara node dan klien browser atau klien API.



Gambar 2.5 Skema Arsitektur Microservices Pada Web E-commerce

Gambar 2.5 menunjukan cara yang mungkin untuk membangun aplikasi web *e-commerce* yang dijelaskan dalam arsitektur *microservices*. Aplikasi dipecah menjadi beberapa layanan kecil yang masing-masing memiliki tujuan sendiri, mengelola pesanan, mengelola produk dan mengelola konten.

Setiap layanan kecil memiliki database sendiri yang berisi data yang dihasilkan atau digunakan oleh layanan tertentu, pada dasarnya kita tidak memiliki database besar melainkan beberapa database kecil. Ini merupakan salah satu keuntungan besar dari *microservices* karena *scaling* database secara horisontal menjadi mungkin. Tentu saja bahwa masalah konsistensi data muncul saat arus kompleks, yang melewati data beberapa layanan, gagal disuatu tempat ditengah pemrosesan. Menjamin konsistensi data sekarang membutuhkan banyak usaha ekstra, sesuatu yang sebelumnya mudah dicapai dengan mengguaan transaksi database.

API Gateway berdiri didepan klien dan menggabungkan API yang ditawarkan oleh semua layanan *microservices* sehingga klaien hanya mengetahui tentang API tersebut dan tidak mengetahui layanan lain dibelakangnya.

2.2.7 Keunggulan Arsitektur Microservices

Microservices memiliki sejumlah keungguan, beberapa keunggulan dari arsitektur ini dapat dijabarkan sebagai berikut:

1. *Microservices* memberikan fleksibilitas bagi developer untuk menentukan tools yang akan digunakan untuk mengimplementasikan sebuah service tanpa harus memaksakan tools tersebut untuk digunakan pada service-service lainnya. Hal ini sangat memungkinkan karena antara satu service dengan service lainnya tidak *tightly coupled* (digabungkan dengan erat). Komunikasi

antara satu service di *microservices* dengan service lainnya umumnya dilakukan via *messaging* atau *web service*. Keduanya bisa diimplementasikan dengan menggunakan tenologi yang berbeda.

2. *Microservices* memungkinkan *team splitting* dilakukan dengan mudah. Satu tim yang menangani sebuah *microservices* bisa dengan mudah memisahkan dari tim lainnya, karena mereka tidak berbagi *code base* dan hanya terikat dengan *interface* yang didefinisikan didalam service-service tersebut.
3. Memisahkan fungsionalitas dalam service-service terpisah menjadi service-service tersebut terisolasi satu sama lain. Dengan kata lain, resiko dan problem dalam satu service akan lebih mudah dilokalisir. Termasuk didalamnya adalah permasalahan dengan *legacy system*(sistem warisan).
4. *Microservices* memungkinkan pembedaan profil satu service dengan service lainnya, terutama yang berhubungan dengan scaling up.

(Ismail Habib, 2014).