# BAB I PENDAHULUAN

Pada bab ini akan dibahas mengenai latar belakang, ruang lingkup masalah, tujuan penelitian, dan manfaat dari penelitian tugas akhir

## 1.1 Latar Belakang

Sistem computer adalah interaksi dari perangkat lunak dan perangkat keras yang membentuk sebuah jaringan elektronik. Tugas dari sebuah sistem adalah menerima input, memproses data input, menyimpan data olahan, dan menampilkan output sebagai bentuk informasi. Dalam penerapannya, kita menyebut sistem aplikasi sebagai program komputer yang bertugas untuk menyelesaikan kebutuhan khusus. Terdapat beberapa tahapan umum dalam mengembangkan sistem aplikasi yaitu perencanaan, analisa, desain, pengembangan, testing, implementasi, dan pemeliharaan [1]. Tahap yang cukup penting dan akan menjadi fokus diskusi disini adalah desain dan pengembangan, yang dimana peran arsitektur perangkat lunak sangat berperan penting untuk menetapkan landasan dasar pengembangan aplikasi dari awal sampai selesai. Hasi dari arsitektur perangkat lunak merupakan struktur-struktur yang menjadikan landasan untuk menentukan keberadaan komponen-komponen perangkat lunak, cara komponen-komponen untuk saling berinteraksi dan organisasi komponen-komponen dalam membentuk perangkat lunak [2]. Secara umum perangkat lunak bekerja untuk pengguna pada *desktop* *browser, mobile browser,* dan aplikasi *browser* lainnya. Aplikasi tersebut mungkin akan menggunakan API (*Application Programming Interface*) sebagai pihak ke 3. Aplikasi juga dapat saling berintegrasi dengan aplikasi lain dengan menggunakan *web service*. Aplikasi bekerja dengan menerima *request* (HTTP *request* dan pesan) dengan menjalankan logika perhitungan, mengakses database, bertukar pesan dengan sistem lain, dan mengembalikan HTML/JSON/XML sebagai respon balikan [4].

IEEE 803:1993 mengelompokkan kebutuhan non-fungsional ke dalam sejumlah kategori kualitas dari suatu perangkat lunak, diantaranya yaitu : ketepatan (*correctness)*, *robustness,* performa, ketersediaan dan kualitas antarmuka (*interface)*, keandalan (*reability),* ketersediaan (*availability*). Ketika skala aplikasi masih kecil dan sedikit data yang digunakan, kebutuhan masih mudah untuk dipenuhi, namun ketika aplikasi semakin besar, akan terjadi masalah yang selain disebabkan oleh data yang banyak, namun juga oleh *load* komputasi yang besar yang berasal dari *multiple user* dari berbagai lokasi.

Model arsitektur yang paling sering digunakan saat ini adalah model monolitik. Arsitektur monolitik merupakan arsitektur yang mudah dimengerti dan di modifikasi karena lebih sederhana implementasinya. Arsitektur ini menggunakan kode sumber dan teknologi yang serupa untuk menjalankan semua tugas-tugasnya. Contoh yang dapat diambil adalah aplikasi Wordpress. Wordpress merupakan contoh yang mudah untuk menggambarkan sebuah aplikasi monolitik, dimana semua fitur seperti *security*, performa, manajemen konten, statistik dan semuanya dibangun dengan menggunakan PHP dan database MySQL dalam kode yang sama [5]. Secara garis besar keunggulan dari arsitektur monolitik dapat dirasakan apabila aplikasi ingin mudah untuk dikembangkan, mudah untuk di *deploy,* dan dapat selalu dipantau pertumbuhan perfomanya [5].

Namun apabila aplikasi semakin besar dan anggota tim semakin banyak, arsitektur monolitik akan menghadapi kekurangan yang semakin lama akan semakin signifikan. Pertama, ketika aplikasi semakin besar, barisan code monolitk akan menyulitkan *developer* terutama yang baru bergabung bersama tim, aplikasi akan sulit dimengerti dan di modifikasi. Akibatnya pertumbuhan aplikasi akan melambat dan terlebih karena sulit dimengerti, kualitas kode akan semakin menurun. Kedua, semakin banyak code yang ditulis, maka akan semakin lambat IDE (Integrated Development Environment) yang digunakan, semakin tidak produktif pula proses *development* yang dilakukan. Hal ini juga berpengaruh pada waktu yang dibutuhkan untuk menjalankan aplikasi pertama kali, serta menjadi semakin sulit untuk memodifikasi aplikasi. Seperti untuk mengubah sebuah komponen, *developer* harus *redeploy* keseluruhan aplikasi. Ketiga, akan sulit untuk membagi team secara fungsionalitas, seperti misalnya membagi tim akunting dan tim inventori. Kedua tim tersebut tidak dapat secara mandiri bekerja sendiri, karena hanya ada 1 aplikasi besar yang mengakibatkan adanya saling ketergantungan.

Model arsitektur Microservice adalah pattern alternatif yang dapat mengatasi keterbatasan dari arsitektur monolitik, model ini mulai muncul ke permukaan di tahun 2015 (Google trend). Menurut artikel yang saya kutip, kebanyakan aplikasi mulai dari arsitektur monolitik, sampai hingga aplikasi itu sulit di kembangkan lagi, kemudian aplikasi dipecah menjadi model microservis, hal itu yang terjadi pada perusahaan besar seperti Netflix dan Amazon [8]. Secara garis besar arsitektur microservis mendefinisikan struktur service yang lebih sempit dengan area fungsi yang saling berkaitan. Tiap servis saling berkomunikasi menggunakan protokol seperti HTTP dan setiap servis bisa mempunyai databasenya sendiri masing-masing. Arsitektur microservis mengubah servis aplikasi menjadi modul yang mandiri, kecil (dibandingkan monolitik), dan setiap servis berjalan sesuai dengan perannya masing-masing dan tidak saling ketergantungan [6]. Dalam artikel yang dijelaskan oleh Chris Richardson, ada banyak pattern model dari arsitektur microservis, tergantung dari aplikasi yang akan dimigrasi.

Dalam contoh kasus penelitian ini, peneliti akan menerapkan arsitektur microservis pada aplikasi monolitik rumah sakit Apentura, aplikasi rumah sakit Apentura mengalami kendala dalam hal perawatan dan pengembangan lanjutan terhadap aplikasinya. Hal ini dapat dirasakan dengan banyaknya data yang semakin banyak dan sulit untuk dipelihara. Kasus lain apabila terjadi kegagalan hardware yang menyebabkan database tidak dapat diakses. Arsitektur aplikasi saat ini tidak dapat menunjang untuk mengatasi masalah tersebut, sehingga segala kegiatan yang berhubungan dengan database akan lumpuh total. Performa kecepatan yang menurun seiring dengan banyaknya *actor* yang mengakses aplikasi yang disebabkan selain karena aplikasi yang terus membesar, juga karena data input dan kebutuhan tiap *actor* yang semakin hari semakin banyak.

Tujuan dari penelitian ini adalah menerapkan arsitektur microservis yang paling cocok untuk kasus aplikasi rumah sakit Apentura, menjelaskan bagaimana tahapan migrasi ke arsitektur microservis, menjelaskan bagaimana analisis yang baik agar dapat menentukan desain arsitektur microservis yang benar, serta membandingkan performa antara arsitektur microservis dan arsitektur monolitik dari segi kecepatan, beban hardware yang dibutuhkan, kemudahan pengembangan, *high availability*, dll.

## 1.2 Rumusan Masalah

Berdasarkan latar belakang yang ada, maka permasalahan yang akan dihadapi yaitu :

1. Bagaimana menentukan pattern microservis dan desain arsitektur yang paling tepat/sesuai untuk aplikasi rumah sakit Apentura.
2. Bagaimana melakukan migrasi dari arsitektur monolitik ke model arsitektur microservice.
3. Bagaimana melakukan perbandingan performa yang dihasilkan dari arsitektur microservis yang baru terhadap arsitektur monolitik.

## 1.3 Batasan Masalah

Berikut adalah batasan masalah yang terdapat pada penelitian ini:

1. Penelitian tidak mencakup semua bagian dari aplikasi rumah sakit Apentura, namun hanya mengambil bagian data pasien, data karyawan, data rekap medis, dan rawat jalan.
2. Segala data yang digunakan dalam penelitian bersifat tertutup karena menyangkut hal privasi dari rumah sakit Apentura.
3. Fokus utama dari penelitian ini adalah berupa analisis perancangan arsitektur dan uji coba perbandingan performa yang dibuat dalam bentuk prototype.

## 1.4 Tujuan Penelitian

Berdasarkan latar belakang dan rumusan masalah, tujuan dari penelitian ini adalah :

1. Menentukan bagaimana tahap yang benar untuk melakukan migrasi dari arsitektur monolitik ke arsitektur microservis.
2. Membandingkan apa saja kelebihan dan kekurangan dari arsitektur microservis dibandingkan dengan arsitektur monolitik.
3. Mengetahui bagaimana analisa untuk membagi servis monolitik menjadi servis microservis yang lebih fokus dan padat.

## 1.5 Manfaat Penelitian

1. Memberikan panduan untuk melakukan migrasi dari arsitektur konfensional ke arsitektur microservis.
2. Memberikan hasil analisa perbandingan performa dari kedua jenis arsitektur.

## 1.6 Metodologi Penelitian

Metodologi atau langkah-langkah yang dilakukan dalam penelitian ini, yaitu:

1. Analisis

Melakukan studi literatur dan analisa melalui wawancara dengan pihak perancang aplikasi monolitik sebelumnya. Data juga dikumpulkan dari jurnal-jurnal, karya ilmiah, dan situs yang memberikan informasi yang menunjang mengenai konsep arsitektur microservis dan tahap-tahap implementasi pada aplikasi monolitik.

1. Identifikasi

Melakukan identifikasi mengenai target-target yang ingin dicapai setelah menerapkan arsitektur baru. Identifikasi ini berguna untuk menentukan perbandingan apa saja yang akan diperhatikan antara arsitektur microservis dan arsitektur monolitik ketika pengujian dilakukan.

1. Perancangan

Perancangan arsitektur microservis meliputi perancangan model dasar (proses bisnis), perancangan sistem basis data, perancangan kelas s*ervice* yang baru dengan konsep microservis, juga perancangan jalur komunikasi dan pertukaran data antara kelas-kelas *service* yang akan dibuat.

1. Implementasi

Melakukan implementasi hasil perancangan dalam bentuk *web service application (server side).* Selanjutnya dibuat aplikasi *client* sederhana untuk input data dan pengujian performa dari arsitektur server yang telah dibuat.

1. Pengujian

Melakukan pengujian terhadap rancangan aplikasi microservis yang baru dan aplikasi monolitik dengan menggunakan data rumah sakit untuk mengetahui hasil kinerja dan perbandingan performa antara kedua jenis arsitektur.

## 1.7 Sistematika Penulisan

Sistematika penulisan digunakan untuk mendokumentasikan laporan penelitian secara utuh dan lengkap. Sistematika yang dibentuk adalah sebagai berikut:

1. Bab I Pendahuluan

Berisi penjelasan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan penelitian, manfaat penelitian, metodologi penelitian, dan sistematika penulisan laporan penelitian.

1. Bab II Landasan Teori

Membahas tentang definisi dari arsitektur microservis, teori-teori pendukung,, dan metode penerapan arsitektur baru yang akan dijadikan landasan dan dipelajari serta dirangkum dari berbagai sumber, seperti buku, karya tulis, jurnal, artikel dari situs ilmiah. Juga pembahasan sekilas mengenai arsitektur konvensional yang digunakan sebelumnya.

1. Bab III Analisa dan Perancangan

Berisi analisa mengenai bagaimana konsep penerapan arsitektur microservice pada aplikasi rumah sakit dan modul-modul apa saja yang akan di migrasi menjadi modul baru untuk diimplementasikan pada tahap selanjutnya. Selanjutnya membahas perancangan aplikasi dari hasil analisa dengan menggunakan arsitektur microservis yang baru.

1. Bab IV Implementasi dan Pengujian

Berisi implementasi dari hasil perancangan aplikasi dalam bentuk perangkat lunak menggunakan teknologi *web service*. Selanjutnya perangkat lunak diuji fungsi dan performanya dari sisi *client* (pengguna).

1. Bab V Kesimpulan dan Saran

Memberikan kesimpulan berdasarkan hasil analisis, perancangan, implementasi dan pengujian, serta evaluasi yang dilakukan, juga saran-saran yang dibutuhkan untuk pengembangan lebih lanjut.

# BAB II LANDASAN TEORI

Pada bab ini akan dijelaskan teori pendukung serta metode yang digunakan untuk mengekstraksi modul-modul yang terdapat pada aplikasi monolitik. Penjelasan teori dimulai dengan pengertian dari arsitektur Microservice, prinsip dan pemodela microservice, membangun arsitktur microservice dari arsitektur monolitik, metode pengembangan aplikasi berbasis Microservice, serta aplikasi rumah sakit Apentura sendiri.

## 2.1 Microservice Architecture

Arsitektur aplikasi yang memiliki struktur hubungan service yang renggang namun kolaboratif. Tiap service memiliki fungsi yang lebih sempit dan saling berhubungan. Tiap service ini saling berkomunikasi menggunakan web service dan dapat dikembangkan dan di *deploy* secara mandiri. Tiap service memiliki database masing-masing yang saling memisahkan data.

## 2.1.1 Definisi Arsitektur Microservice

Aritektur microservice pertama kali muncul untuk memenuhi kebutuhan untuk menunjukan bagaimana sebuah aplikasi dapat lebih efektif dan efisien untuk masuk kedalam tahap *production*, juga untuk menunjukan bagaimana *development* yang lebih baik dengan memberikan kemampuan kepada mesin untuk saling berkomunikasi. Microservice juga termasuk ke dalam perancangan insfrastrktur mesin sampai skala yang dibutuhkan. Banyak organisasi telah membuktikan dengan berpindah ke arsitektur microservice, aplikasi mereka menjadi lebih cepat dan berani untuk menggunakan teknologi yang baru. Microservice memberikan *developer* kebebasan untuk bereaksi dan mengambil keputusan yang berbeda, memberikan respon yang lebih cepat atas segala kebutuhan dari pengguna aplikasi. [9]

## 2.1.2 Prinsip Pendekatan Arsitektur Microservice

Terdapat beberapa tahapan dalam pendekatan dalam arsitektur mikroservis yang menjadikan desain system yang baik, pendekatan ini berguna untuk mendefinisikan prinsip dan petunjuk yang bergantung pada gol yang kita tuju, tahapan pendekatan tersebut yaitu :

1. ***Strategic Goals*.** Strategic goals harus memberikan arahan kemana perusahaan ingin beranjak dan bagaimana memenuhi kebutuhan konsumen. Bahasan ini harus berisi tujuan tertinggi dan tidak membahas teknologi sama sekali. Goals ini bisa dibahas di level perusahaan atau juga di level divisi. Kuncinya adalah untuk membuat kemana arah organisasi akan bergerak. [9].
2. ***Principles*.** Principles adalah aturan yang harus dibuat agar dapat memenuhi goals, prinsip ini kadang berubah sesuai dengan kondisi. Misalnya apabila strategic goals perusahaan adalah untuk mengurangi waktu pengiriman barang-barang baru, maka organisasi terebut akan mendefinisikan prinsip yang mengatakan bahwa tim pengiriman mempunyai kontrol penuh terhadap *lifecycle* produk mereka untuk dikirimkan kapanpun produk siap. Namun apabila goals adalah untuk mengembangkan pertumbuhan produk dengan cepat di sebuah negara, maka organisasi akan memutuskan untuk mengimplementasi prinsip bahwa semua system harus bisa bekerja secara portable agar dapat di *deploy* secara local dan memastikan bahwa data akurat. Prinsip ini juga jangan terlalu banyak, kurang dari 10 adalah angka yang baik, karena semakin banyak prinsip akan beresiko menjadikan aturan-aturan tersebut saling bentrok satu sama lain. [9]
3. ***Practices*.** Tahap ke tiga adalah untuk memastikan semua prinsip telah dilakukan. *Practices* disini adalah sebuah detail set, bagaimana untuk melakukan task-task agar goals dapat dicapai sesuai dengan aturan yang ada. Tahap ini termasuk dengan spesifikasi teknologi, dan harus cukup sedetail mungkin agar semua *developer* dapat paham. *Practices* dapat termasuk petunjuk bagaimana *coding*. Sesuai dengan sifat naturalnya, *practices* akan lebih sering berubah dibandingkan dengan principal di tahap ke 2. [9]
4. ***Combining Principles and Practices.*** Ide dari point terakhir ini adalah ketika system berevolusi dengan ide baru, organisasi tetap siap dengan segala detail yang dibutuhkan agar semua orang tahu bagaimana mengimplementasi ide baru tersebut. Terdengar mudah untuk dilakukan di lingkup yang kecil, namun untuk lingkup besar, bisa terdapat perbedaan antara teknologi dengan praktek yang dilakukan. Misalnya tim .NET akan mempunyai set *practices* yang berbeda dengan tim Java. [9]

## 2.1.3 Memodelkan *Services*

Setelah pengertian umum mengenai arsitektur microservis, pada bagian ini akan dijelaskan bagaimana cara berfikir dengan batasan-batasan microservice yang akan memaksimalkan semua potensinya. Dalam bab ini peneliti menginginkan pembaca fokus terhadap dua konsep kunci microservice, yaitu *loose coupling* dan *high cohesion*.

***Loose Coupling.*** Ketika service telah *loosely coupled*, perubahan yang dilakukan terhadap satu service tidak akan mengakibatkan perubahan pada service yang lain. Point disini adalah bagaimana microservice dapat melakukan perubahan pada satu service dan melakukan *deploy* tanpa harus melakukan perubahan apapun pada sistem. Namun sebuah sistem dapat memiliki kebutuhan berkomunikasi antar service, hal ini mengakibatkan arsitek harus membatasi limit panggilan dari satu service terhadap service yang lain, karena selain dapat menyebabkan masalah performa, hal ini pula dapat mengakibatkan terjadinya *tight coupling.*

***High Cohesion***. Model microservice menginginkan sifat-sifat yang berkaitan untuk berada di satu wadah, dan yang tidak berkaitan ditempatkan di wadah yang lain, karena apabila ada perubahan yang terjadi, hanya satu wadah tersebut yang akan berubah dan perubahan dapat langsung di implementasikan dengan cepat. Apabila service dibuat terlalu tercecer, maka akan menyebabkan perubahan di banyak tempat dan akan membuang banyak waktu. Point yang diinginkan adalah menempatkan service dengan sifat yang mirip di satu wadah, namun tetap berkomunikasi dengan wadah lain selonggar mungkin.

## 2.1.4 Shared dan Hidden Model

Dalam buku yang dijelaskan Sam Newman, penulis mengambil contoh sebuah departemen keuangan dan departemen *warehouse* di sebuah organisasi untuk menjelaskan tentang shared dan hidden model. Kedua departemen mempunyai *interface* yang berbeda ketika ditampilkan. Departemen keuangan tidak perlu tahu segala detail di *warehouse*. Namun walau begitu tetap ada data yang dibutuhkan seperti misalnya stok barang agar mendapatkan perhitungan terbaru. Pada model microservice maka ke dua modul ini akan dibuat terpisah. Berikut penggambarannya :



Figure 1. Model pembagian dari departemen keuangan dan warehouse

Untuk dapat menjalankan alur informasi, pegawai keuangan membutuhkan data stok. *Stock item* menjadi *shared model* antara dua departemen. Perlu diingat bahwa tidak semua data *warehouse* harus diperlihatkan di keuangan, jadi terdapat representasi internal dan representasi external yang diperlihatkan. Metode *shared model* ini akan dibicarakan dalam lingkup *web service* REST. Memikirkan secara matang model apa saja yang harus di *share*, dapat mencegah terjadinya *tight coupling* (berlawanan dengan yang diinginkan).

Desain diatas adalah contoh modul yang harus dibuat agar model microservcie dapat direpresentasikan dengan jelas. Desain ini sangat mempermudah proses perpindahan dari monolitik dan menyakinkan bahwa desain microservice telah *loosely coupled* dan *stongly cohesive*

## 2.2 Integrasi

Mengintegrasikan dengan benar merupakan tahap yang paling penting, memungkinkan perubahan yang signifikan dengan tingkat kemandirian aplikasi yang tinggi. Pada bagian sebelumnya sempat dibahas mengenai bagaimana microservice dapat berbicara antara satu model dengan model lainnya, pada bagian awal integrasi, akan dibahas teknologi apakah yang tepat sebagai media komunikasi. Media komunikasi disini bisa meliputi SOAP, XML-RPC, REST, atau *protocol buffers.* Dalam tahap integrasi ini ada beberapa point penting yang harus dianalisis sebelum memilih teknologi yang digunakan dan mengimplementasikannya.

1. **Menjaga teknologi API agar tetap agnostik**. IT industri adalah berubah dengan sangat cepat, *tools* baru, *framework* dan bahasa baru, serta ide-ide implementasi yang selalu berkembang. Hal inilah yang menjadi pertimbangan agar memastikan bahwa API inisial harus dapat digunakan terus menerus ketika mengimplementasikan microservice.
2. **Hindari perubahan major pada aplikasi**. Perubahan arsitektur dapat mengakibatkan perubahan pada bagian-bagian aplikasi yang lainnya, pemilihan teknologi yang tepat bertujuan agar perubahan ini terjadi sekecil mungkin.
3. **Buat service sederhana untuk dipakai**. Arsitektur microservice yang baru harus cepat beradaptasi dengan penggunanya, maka dari itu modul service harus bersifat *user-friendly*.

## 2.2.1 Shared Database

Database menjadi hal yang paling penting untuk diintegrasikan. Dalam kasus arsitektur biasa, apabila service menginginkan data dari service lain, maka sistem akan mengambil data tersebut dari database, sama hal nya apabla data diubah, maka sistem akan langsung mengubah database. *Life cycle* seperti ini sangat simpel dan sangat cepat sehingga sampai saat ini dipakai oleh banyak sistem.

Contoh dalam gambar 2, *customer* yang akan melakukan registrasi akan langsung melakukan SQL ke database, juga aplikasi *call center* yang menampulkan dan mengubah data akan langsung melakukan SQL ke database, begitu pula dengan informasi *update warehouse* mengenai pesanan konsumen, akan melakukan *query* pada database. Ini adalah contoh pattern yang sangat umum, namun penuh dengan kesulitan



Figure 2. Contoh pemodelan database umum

Pertama, model ini mengizinkan langsung pihak luar untuk merubah data internal. Struktur data yang di simpan di DB dipakai oleh semua *user*, apabila terjadi perubahan pada DB, maka semua *user* akan terkena dampaknya. Database menjadi sangat besar, dan *shared* API menjadi rapuh. Apabila akan terjadi perubahan, misalnya perubahan table customer di database, maka harus sangat berhati-hati agar *schema* yang dipakai oleh service lain tidak rusak. Hal ini membutuhkan usaha testing regresi yang besar. Hal ini melanggar konsep *loose coupling.*

Kedua, semua *client* menjadi terikat dengan sebuah teknologi spesifik. Mungkin saat ini database berjalan dengan baik dengan menggunakan *relational* database, namun bagaimana bila seiring berjalannya waktu, performa untuk menyimpan data lebih baik menggunakan *nonrelational* database? *Client* menjadi terikat dengan model implementasi. Hal ini melanggar konsep *cohesion*.

Ketika membangun sebuah rancangan microservice, prinsip *loose coupling* dan *high cohesion* adalah yang utama. Database yang terintegrasi baik untuk berbagidata, namun tidak baik untuk berbagi *behavior*. Maka dari itu pada setiap service memiliki gaya integrasi databasenya masing-masing.

## 2.2.2 Memilih Remote Procedure Calls (RPS) yang Tepat

Dengan memiliki beberapa database dan modul service yang berbeda-beda, timbul sebuah masalah yang berkaitan dengan pertukaran informasi yang berasal dari banyak service. *Remote Procedure Calls* adalah protocol yang menyediakan teknik komunikasi dengan service yang berbeda lokasi. Kebanyakan teknologi RPC ini menggunakan kode biner sebagai format pertukaran data, ada pula yang menggunakan format pesan XML seperti SOAP. Implementasi RPC ini berguna untuk mendapatkan data dengan sangat cepat yang dikirimkan melalui jaringan, hal yang menjadi keuntungan utama dari RPC adalah kemudahan penggunaannya. Contoh RPC lain yang menjadi fokus disini adalah Representational State Transfer (REST), terdapat *plus minus* dari penggunaan REST, namun point berikutnya akan menjelaskan mengapa REST sangat membantu masalah integrasi di microservice.

**Representational State Transfer (REST).** REST adalah standar arsitektur web yang kebanyakan menggunakan protokol HTTP. HTTP sendiri mempunyai kemampuan yang sangat cocok untuk REST, salah satunya HTTP faham apa yang harus dilakukan apabila menerima perintah GET, POST, PUT dari REST. Kelebihan penting yang dimiliki REST adalah pengguna bisa menghindari kontak langsung dari pengguna dengan server secara langsung. Konsep ini kemudian disebut sebagai *hypermedia as the engine of application state* (HATEOAS)*.* *Hypermedia* adalah konsep dimana sebuah konten mempunyai *link* yang berhubungan dengan konten lainnya yang bisa berupa berbagai format (text, gambar, suara). Ide dari HATEOAS adalah *client* berhubungan dengan server hanya dengan menggunakan *link* yang telah disediakan.

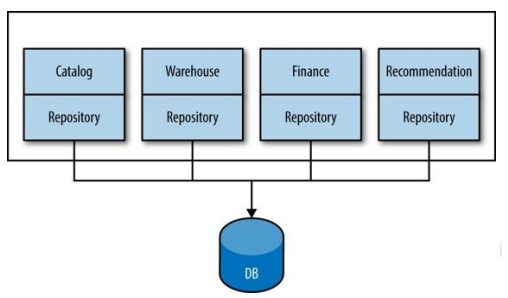
Figure 3. Contoh pemanfaatan REST sebagai media user-server

Misalnya seperti gambar *figure 3.* Ketika pengguna ingin mengubah data *account*, maka pengguna akan mengirimkan *request* pada *account service*, *account service* dengan logic yang dimilikinya akan menentukan apakah *request* tersebut dapat diterima. *Account service* disini menjaga semua interaksi yang berhubungan dengan data *account* itu sendiri. *Client* tidak perlu tahu dan tidak perlu beradaptasi apabila terjadi perubahan pada server. *Client* akan merasakan perubahan hanya apabila terjadi perubahan sifat atau ketika hilangnya kontrol yang merepresentasikan *account*. Format data yang dikirimkan REST di HTTP dapat beragam, namun yang paling populer adalah format JSON, karena JSON mudah dimengerti dan mudah dikonsumsi langsung. REST pada HTTP sangat baik untuk diimplementasikan pada interaksi *sevice-to-service.* [9]

## 2.3 Memisahkan Monolitik

Setelah menjelaskan teori microservice yang baik, bagaimana mengatasi masalah apabila aplikasi saat ini menggunakan arsitektur monolitik dan telah berkembang sangat besar dan menjadi sangat beresiko untuk dipecah. Point ini akan menjelaskan contoh *decomposing* aplikasi monolitik tanpa harus merombak besar aplikasi. Untuk dapat melakukan hal tersebut, *decomposing* akan fokus pada inti dari sistem, yaitu database. Tahapan untuk memecah database adalah :

1. **Memecah Entitas.** Hal pertama yang harus dilakukan adalah membuat *packages* yang merepresentasikan entitas, lalu memindahkan semua *code* ke dalam *packages* tersebut. Proses ini sering juga disebut dengan mengubah *code* menjadi *domain-oriented packages*. Hal ini dapat dilakukan dengan me*refactoring* menggunakan IDE modern. Seiring berjalannya waktu, *developer* akan tahu *code* mana yang bisa atau tidak bisa dipakai lagi.
2. **Pemisahan Database.** Pada bab sebelumnya, penulis telah membahas kesulitan apabila hanya terdapat 1 database untuk meng*-handle* banyak service, artinya *developer* harus menemukan celah pada database agar bisa dipisahkan dengan baik. Tahap untuk memisahkan database antara lain:
   1. **Melihat ketergantungan objek.** Langkah pertama adalah melihat bagian apa yang melakukan *read* dan *write* pada database. Metode yang paling umum digunakan adalah menggunakan bantuan *framework* seperti Hibernate yang memetakan objek dengan database. Dengan bantuan pemetaan seperti ini, akan sangat mempermudah pemahaman hubungan antara tables, hal inipun berlaku untuk tables yang mempunyai relasi dengan tables lain.Seperti contoh pada figure 4 :



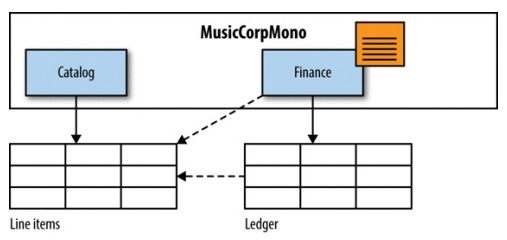
* 1. **Melepas Ketergantungan Foreign-Key.** Sam Newman dalam bukunya memberikan contoh kasus dari 2 buah model yang saling berinteraksi, model Catalog dan tabel Finance. Model catalog merepresentasikan tabel *line items*, dan model finance merepresentasikan tabel *ledger*. Setiap akhir bulan, perusahaan harus memberikan laporan penjualan kepada semua staf. Untuk melakukan hal ini, laporan di finance harus mengambil data dari tabel *line items*, hal ini mengakibatkan adanya foreign key dari tabel *ledger* kepada tabel *line item.*

Figure 4. Relasi Foreign Key

Hal yang pertama harus dilakukan adalah menghentikan finance untuk mengambil data langsung dari tabel *line items*, karena tabel ini dimiliki oleh model catalog dan patern microservice tidak ingin adanya integrase database karena catalog dan finance akan dipecah menjadi servicenya masing-masing. Solusi untuk mencegah pengambilan data langsung dari tabel *line item* adalah dengan membuat API catalog untuk dipanggil di *package* finance, metode pemanggilan API inilah yang akan menjadi peranatara komunikasi antar tabel. Ilustrasi pada gambar 5.

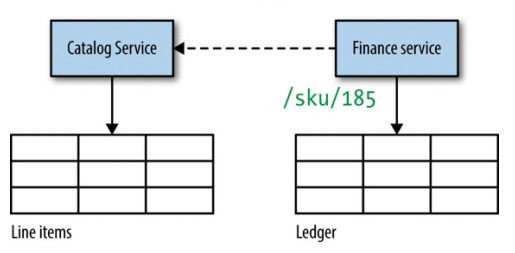


Figure 5. Mengganti relasi foreign key dengan pemanggilan API

Pemanggilan API pun dapat melibatkan lebih dari 1 tabel dari beberapa service. Dengan memisahkan tabel, maka hal ini pula meningkatkan point *availability* dari sistem, karena masing-masing tabel akan tetap bisa digunakan apabila tabel relasinya mengalami kerusakan atau bahkan hilang. Misalnya dalam tabel *ledger* terdapat ID untuk *line item*, apabila tabel *line items* dhilangkan, hal ini tidak akan menyebabkan tabel *ledger* menjadi tidak bisa diakses, karena tidak ada keterikatan relasi langsung diantara 2 tabel itu saat ini.

* 1. ***Shared Static Data.*** Dalam beberapa kasus akan ditemukan sebuah tabel yang digunakan secara bersamaan untuk beberapa service, contoh kasus seperti gambar 6.

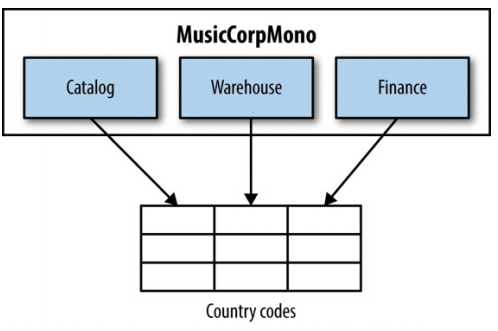
Tabel *country codes* yang digunakan *catalog, warehouse, dan finance.* 

Figure 6. Data yang digunakan bersamaan

Terdapat 3 solusi untuk mengatasi masalah ini. Pertama adalah dengan menduplikat tabel untuk digunakandalam ketiga *packages*. Namun hal ini akan menimbulkan masalah baru berkaitan dengan data konsistensi, misalnya ketika terjadi perubahan untuk *record* kode Australia di tabel 1 namun tidak disemua tabel. Pilihan kedua adalah tetap menjadikan tabel *country code* sebagai tabel *relational* dan pemanggilan data tetap menggunakan *coding* manual. Ketiga adalah menjadikan tabel *country codes* sebagai service dan database mandiri, lalu pengambilan data dilakukan menggunakan API yang disediakan.

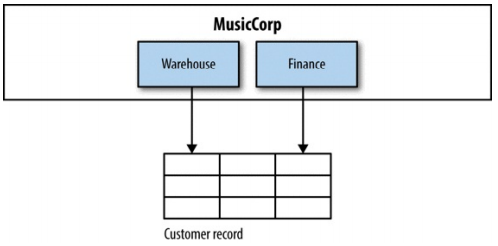
* 1. ***Shared Data.*** Kasus yang lebih kompleks apabila bukan hanya *read* data, namun juga *write dan update*. Misalnya bagian *finance* membutuhkan data *customers* untuk melihat pembayaran, sementara itu *warehouse* melakukan update *records* terhadap status pesanan apabila pesanan *customer* telah sampai. Seperti ilustrasi pada gambar 7. 

Figure 7. Shared data

*Finance* dan *warehouse* keduanya melakukan *read, write,* dan *upate* pada tabel yang sama. Hal ini akan sering ditemukan, dan ini terjadi karena adanya *miss* dalam merepresentasi data untuk tabel relasi. Cara menyelesaikan masalah ini adalah dengan membuat *package* model dari tabel *customer* itu sendiri, seperti pemodelan pada gambar 8. Lalu menggunakan API untuk mengakses data *customer* untuk digunakan pada *packages* lain.

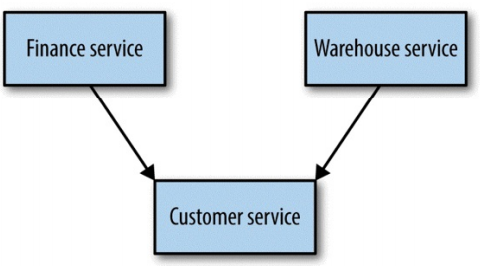


Figure 8. Merepresentasikan tabel customer menjadi model service

* 1. ***Shared Tables.*** Pemecahan database dilakukan sampai sekecil mungkin sampai memungkinkan setiap service hanya memiliki 1 buah tabel. Seperti gambar 9. *Catalog* butuh untuk menyimpan nama dan harga dari barang yang akan dijual, *warehouse* akan meng*-update* stok inventori barang. Kedua model ini menyimpan dan mengubah data pada tabel yang sama.

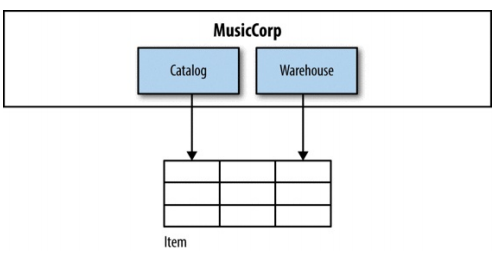


Figure 9. Tabel yang digunakan oleh model yang berbeda

Solusinya adalah membagi tabel menjadi 2 seperti pada gambar 10, masing-masing tabel digunakan oleh modelnya sendiri.

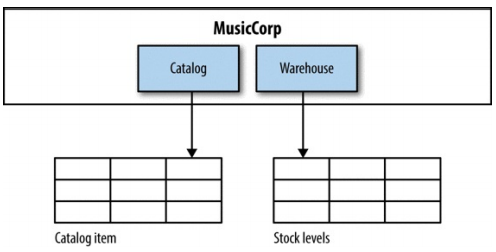


Figure 10. Membagi tabel untuk masing-masing model

[1] Hartono, M. (7). Langkah mudah membangun sistem informasi ERP. *Jakarta: PT. Elex Media Komputindo*.

[2] Pangera, A. A., & Ariyus, D. ARSITEKTUR SISTEM OPERASI.

[3] Pradhananga, Y., & Rajarajeswari, P. (2017). Tiarrah Computing: The Next Generation of Computing. *International Journal of Informatics and Communication Technology (IJ-ICT)*, *6*(2), 129-138.

[4] Chris Richardson (2017). Pattern: Monolithic Architecture.

[5] Polychniatis, T., van der Rijnst, S., van Vliet, R., & Wirken, G. (2013). Software architecture WordPress. *Utrecht, NE: Utrecht University. URL: http://www. cs. uu. nl/wiki/pub/Swa/CourseLiterature/arch-E. pdf [June 24, 2015]*.

[6] Chris Richardson (2017). Pattern: Microservice Architecture

[7] Mazzara, M., Mustafin, R., Safina, L., & Lanese, I. (2016). Towards microservices and beyond: An incoming paradigm shift in distributed computing. *arXiv preprint arXiv:1610.01778*.

[8] Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116-116.

[9] Sam Newman (2015). Building Microservices. O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

[10] Susan J. Fowler(2017). Microservices in Production. O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472