# 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, yaitu: ketepatan (*correctness)*, *robustness,* performa, ketersediaan dan kualitas antarmuka (*interface)*, keandalan (*reability),* ketersediaan (*availability*) [8]. 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 dimodifikasi 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 monolitik 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 [7].

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 Thones, J. (2015) kebanyakan aplikasi mulai dari arsitektur monolitik, sampai hingga aplikasi itu sulit di kembangkan lagi, kemudian aplikasi dipecah menjadi model microservice, hal itu yang terjadi pada perusahaan besar seperti Netflix dan Amazon [8]. Secara garis besar arsitektur microservice 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 microservice 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 microservice, tergantung dari aplikasi yang akan dimigrasi.

Dalam contoh kasus penelitian ini, peneliti akan menerapkan arsitektur microservice pada aplikasi monolitik rumah sakit Apertura. Aplikasi rumah sakit Apertura 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 microservice yang paling cocok untuk kasus aplikasi rumah sakit Apertura, menjelaskan bagaimana tahapan migrasi ke arsitektur microservice, menjelaskan bagaimana analisis yang baik agar dapat menentukan desain arsitektur microservice yang benar, serta membandingkan performa antara arsitektur microservice dan arsitektur monolitik dari segi kecepatan, beban hardware yang dibutuhkan, kemudahan pengembangan, *high availability*, dll.

## 1.2 Rumusan Masalah

Berdasarkan latar belakang yang ada, rumusah masalah dalam penelitian ini yaitu:

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

## 1.3 Batasan Masalah

Berikut adalah batasan masalah yang terdapat pada penelitian ini:

1. Penelitian berfokus pada proses rawat jalan dan tidak melibatkan rawat inap.
2. Sebagian data pasien dan HR yang digunakan dalam penelitian bersifat tertutup karena menyangkut hal privasi dari rumah sakit Apertura.
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 microservice.
2. Membandingkan apa saja kelebihan dan kekurangan dari arsitektur microservice dibandingkan dengan arsitektur monolitik.
3. Mengetahui bagaimana analisa untuk membagi servis monolitik menjadi servis microservice yang lebih fokus dan padat.

## 1.5 Manfaat Penelitian

1. Memberikan panduan untuk melakukan migrasi dari arsitektur konfensional ke arsitektur microservice.
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 microservice 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 microservice dan arsitektur monolitik ketika pengujian dilakukan.

1. Perancangan

Perancangan arsitektur microservice meliputi perancangan model dasar (proses bisnis), perancangan sistem basis data, perancangan kelas s*ervice* yang baru dengan konsep microservice, 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 microservice 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 microservice, 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 microservice 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 Apertura 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 dan menunjukan bagaimana sebuah aplikasi dapat lebih efektif dalam tahap *production*, juga menunjukan bagaimana cara *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* 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 Konsep Microservice

Setelah pengertian umum mengenai arsitektur microservice, pada bagian ini akan dijelaskan bagaimana cara berfikir dengan batasan-batasan microservice yang akan memaksimalkan semua potensinya. Dalam point 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. Prinsip ini menekankan 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.* [9]

***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. [9]

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 2‑1. 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. Desain diatas memperlihatkan konsep *loose coupling* dan *high cohesion* yang digambarkan menjadi sebuah modul. Desain seperti ini sangat mempermudah proses perpindahan dari monolitik dan menyakinkan bahwa desain microservice telah *loosely coupled* dan *stongly cohesive* [9].

## 2.2 Integrasi Teknologi

Mengintegrasikan dengan benar merupakan tahap yang paling penting, memungkinkan perubahan yang signifikan dengan tingkat kemandirian aplikasi yang tinggi. 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*.

Seperti yang dikutip dari *website* Chris Richardson, microservice merupakan sekumpulan teknologi yang saling bekerjasama. Teknologi tersebut tidak dibatasi oleh sebuah wadah tertentu, namun saling terpisah yang menyebabkan luasnya pemilihan teknologi yang akan digunakan. Point berikutnya akan menjelaskan beberapa pilihan teknologi yang baik yang dapat diimplementasikan. Mulai dari lapisan paling dalam, yaitu managemen data, pembagian service, metode berkomunikasi antar service, sampai dengan API *gateway* yang akan digunakan oleh *client.* [9]

## 2.2.1 Manajemen Data

Bab ini akan membahas bagaimana penyimpanan data pada arsitektur monolitik dan apa kelemahannya. Kemudian dilanjutkan dengan penjelasan dan pembahasan metode penyimpanan data yang baik dan sesuai dengan arsitektur microservice

## 2.2.1.1 Model Penyimpanan Data Arsitektur Monolitik

Pada arsitektur biasa, umumnya database disimpan dalam 1 tempat dan terdiri dari beberapa table. Ketika terjadi permintaan untuk membaca data, maka sistem akan mengambil data tersebut dari database, sama halnya apabila 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 2-1, *customer* yang akan melakukan registrasi akan melakukan *querry* ke database, juga aplikasi *call center* yang menampilkan dan mengubah data akan langsung melakukan *querry* ke database, begitu pula dengan informasi *update warehouse* mengenai pesanan konsumen, akan melakukan *query* pada database. Ini adalah contoh pattern yang sangat umum, namun banyak kelemahan dari pattern database ini. [9]



Figure 2‑2. Contoh pemodelan database umum

Pertama, model ini mengizinkan langsung pihak luar untuk mengubah 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.* [9]

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*.

## 2.2.1.2 Model Penyimpanan Data Arsitektur Micoservice

Dalam perancangan arsitektur microservice, terdapat 2 pattern database ditinjau dari pembagian database tersebut. Pattern pertama adalah *shared database* dan yang kedua adalah *database per service.* Untuk contoh kedua pattern, penulis Chris Richardson memberikan contoh dengan menggunakan modul *customer* dan modul *order.* Hubungan kedua modul tersebut terjadi ketika ada transaksi baru, dimana modul *order* harus memastikan bahwa jumlah pesanan yang baru tidak melebihi limit kredit yang dimiliki *customer*. Relasi kedua modul itu dapat dilihat dari gambar dibawah.[6]

Figure 2‑3. Shared Database

***Shared database.*** Model yang pertama adalah database yang di *share* untuk diakses oleh beberapa service. Modul *order* bisa langsung mengakses table *customer* untuk mendapatkan limit kredit *customer* tersebut. Model ini dapat menjadi pilihan ketika *developer* ingin model yang familiar dan tegas untuk menjaga konsistensi data. Model *shared database* memiliki beberapa kelemahan, antara lain :

1. *Development time coupling.* Modul *order* harus tahu apabila terjadi perubahan pada *schema customer,* hal ini menjadikan kedua modul menjadi memiliki ketergantungan. Hal ini akan memperlambat proses *development.*
2. *Runtime coupling.* Karena beberapa service dapat mengakses database yang sama, terdapat potensi mengganggu proses yang lain, misalnya ketika modul *customer* sedang melakukan *update* terhadap *customer* dengan id 234 untuk merubah kredit limit, maka modul *order* harus menunggu sampai *update customer* selesai dilakukan karena *customer* dengan id 234 akan di *block* sementara waktu.

***Database per service.*** Pattern database yang kedua menjadikan sebuah table menjadi *private* hanya untuk 1 buah service dan hanya dapat diakses via API, service lain tidak bisa mengakses database tersebut. Kelebihan dari model ini adalah menjadikan service lebih *loose coupled*, tingkat ketergantungan antar service rendah. Tiap service pun dapat memiliki database yang cocok untuk dirinya sendiri. Namun model database ini memiliki beberapa kesulitan, antara lain :

1. Mengimplementasikan proses bisnis yang melibatkan banyak service menjad lebih sulit, dan lebih baik dihindari karena akan menemukan kesulitan di integritas data, terutama database modern (NoSQL). Solusi terbaik adalah dengan menggunakan konsep SAGA pattern (dibahas pada point berikutnya), yang menjalankan *event* ketika ada perubahan data. Service lain yang men-*subscribe event* tersebut akan merespon dan melakukan *update* juga. [6]
2. Mengimplementasi *query* yang menggabungkan data dari banyak database akan lebih sulit. [6]

## 2.2.2 Dekomposisi Modul

Tujuan dari dekomposisi modul ini adalah menemukan service terkecil penyusun aplikasi. Dengan memisahkan modul menjadi sebuah service tunggal, menjadikan aplikasi lebih mandiri dan mudah untuk di koordinasikan dalam tim, yang dapat mempercepat proses *development*. Tujuan yang lebih besar lagi adalah menghindari perubahan besar pada aplikasi ketika ada proses bisnis yang berubah. Dengan mendekomposisi modul, akan membantu untuk memastikan hanya ada sebuah service saja yang akan terkena dampaknya.

Tantangan yang didapat ketika hendak melakukan dekomposisi modul adalah:

* Rancangan arsitektur service yang baru harus stabil.
* Sebuah service harus terdiri dari susunan *functions* yang erat fungsinya.
* Service harus memastikan tidak melanggar *Common Closure Principle,* yaitu apabila terjadi perubahan tidak akan melibatkan *package* lain.
* Setiap service sebagai API yang terenkapsulasi dari pengguna. Segala perubahan tidak boleh memberikan dampak secara langsung pada pengguna.
* Sebuah service harus cukup kecil untuk ditangani oleh tim yang terdiri dari 6-10 orang.
* Service harus bisa di tes, dan setiap tim harus dapat melakukan proses *develop* dan *deploy* tanpa banyak interaksi dengan tim lain.

Terdapat 2 metode dalam mendekomposisi modul aplikasi, yaitu berdasarkan subdomain dan berdasarkan proses bisnis yang dilakukan. [6]

**Dekomposisi berdasarkan subdomain.** Apabila aplikasi yang dibentuk terpisah berdasarkan *domain-driven design* (DDD) atau disebut juga subdomain, maka metode ini dapat lebih mudah digunakan. DDD memisahkan aplikasi berdasarkan kebutuhannya dan membentuk sebuah subdomain sendiri, setiap subdomain bertanggung jawab untuk menangani proses bisnis yang berbeda.Subdomain dapat diklasifikasikan sebagai berikut :

* *Core –* kunci pembeda dari proses bisnis dan menjadi bagian yang sangat penting dari sebuah aplikasi.
* *Supporting –* berhubungan dengan proses bisnis namun bukan menjadi pembeda.
* *Generic –* tidak berhubungan dengan proses bisnis dan biasanya hanya menjadi proses pendukung saja.

Kelebihan dari metode ini antara lain:

* Arsitektur service baru yang stabil, karena subdomain sendiri secara relatifsudah stabil.
* Service yang dibentuk akan cenderung memenuhi *loosely coupled* dan *cohesive*.

Masalah yang dihadapi dari metode ini antara lain:

* Sulit diterapkan untuk aplikasi yang tidak *domain-driven* atau aplikasi yang memiliki keterikatan modul yang kuat.

**Dekomposisi berdasarkan proses bisnis.** Pattern alternatif yang lain adalah mendefinisikan service berdasarkan kemampuan bisnis. Kemammpuan bisnis adalah proses yang dilakukan untuk menghasilkan sebuah nilai. Kemampuan bisnis biasanya berhubungan dengan objek bisnis. Misalnya *order management* bertanggung jawab untuk *orders*, *customer management* bertanggung jawab untuk *customer.* Kemampuan bisnis biasanya tersusun menjadi hirarki multi level, misalnya untuk aplikasi perusahaan memiliki *product/service development*, setelah itu ada *product/service delivery*, lalu diikuti dengan *aftersale service*. [6]

Kelebihan dari metode ini antara lain:

* Arsitektur service baru yang stabil, karena kemampuan bisnis pun relatif sudah stabil.
* Service yang dibentuk akan memenuhi konsep *loosely coupled* dan *cohesive*.

Masalah yang akan dihadapi dari metode ini antara lain:

* Sulit untuk mengidenfitikasi kemampuan bisnis aplikasi. Identifikasi pembentukan service membutuhkan pemahaman dari proses bisnis, maka harus dilakukan analisa mendalam dari tujuan perusahaan, struktur, dan cakupan area. Analisa dapat dilakukan pertama-tama dari struktur organisasi, karena perbedaan grup dalam organisasi berkaitan dengan kemampuan bisnis dari grup tersebut.

## 2.2.3 *Service Deployment*

Setelah mengindentifikasi dan membentuk service, tahap selanjutnya yang akan dilakukan adalah melakukan *deployment* service untuk nantinya digunakan. Tiap service sendiri bisa ditulis menggunakan bahasa dan framework yang berbeda-beda, namun secara mandiri harus *deployable* dan *scalable*. Dalam beberapa kasus, akan terdapat set service yang harus terisolasi dari service lainnya. Kebutuhan lainnya adalah adanya keperluan untuk dapat memantau penggunaan *resources* (CPU dan memori) yang digunakan oleh service dan memantau dengan mudah sifat dan kegunaan dari service tersebut. Semua kebutuhan ini juga harus sebanding dengan biaya yang dikeluarkan.

Terdapat beberapa metode *deployment* yang dapat dilakukan, tiap metode cocok digunakan sesuai dengan kebutuhan dari *deployment* itu sendiri. Point berikutnya akan menjelaskan metode-metode *deployment* service.

1. ***Multiple service instances per host.***

Metode *deployment* ini cocok digunakan apabila service yang terbentuk tidak banyak dan adanya kebutuhan untuk menghemat penggunaan sumber daya. Set service yang terbentuk disimpan dalam sebuah host (fisik atau *virtual machine*). Terdapat 2 cara dalam melakukan *deploy* set service dalam sebuah host. Pertama, *deploy* set-set service tersebut sebagai JVM, misalnya dengan Tomcat atau Jetty untuk sebuah set service. Kedua adalah melakukan *deploy* semua set service dalam sebuah JVM yang sama, mislanya sebagai web aplikasi. [6]

Metode *deployment* ini juga terdapat beberapa kelemahan, antara lain:

* Beresiko terjadi konflik terhadap kebutuhan sumber daya (memori dan CPU).
* Beresiko terjadi konflik versi *dependency.*
* Sulit untuk membatasi konsumsi sumber daya yang digunakan sebuah service (berhubungan dengan point pertama).
* Apabila banyak set service di *deploy* dalam sebuah mesin yang sama, maka akan sulit untuk memonitor konsumsi sumber daya dari tiap service, karena sulit untuk melakukan isolasi terhadap service.

1. ***Single service instance per host.***

Pilihan yang kedua adalah setiap service memiliki hostnya sendiri. Dengan menerapkan pattern ini, setiap service akan terisolasi dari yang lainnya, serta tidak akan ada resiko konflik akan sumber daya dan *dependency.* Setiap service lebih mudah untuk dimonitor dan dikelola. Namun kekurangan dari patern ini apabila dibandingkan dengan *multiple service per host* adalah kurangnya efisiensi penggunaan sumber daya, karena bisa jadi akan ada banyak host yang digunakan. [9]

Figure 2-5. One Service per Host

Figure 2‑4. Multiple Service per Host

1. ***Service instance per container.***

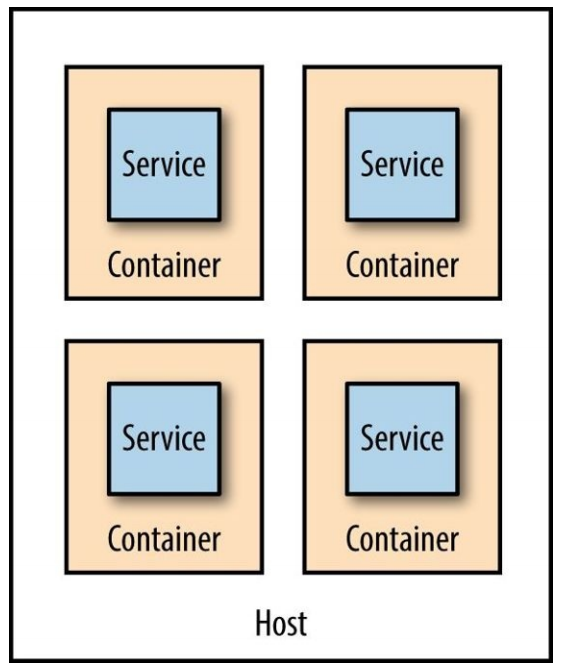
Apabila masing-masing service dibuat dengan bahasa atau framework yang berbeda, maka proses *deployment* dari setiap service akan berbeda-beda pula. Dengan menggunakan *container,* semua detail teknologi yang digunakan oleh setiap service akan dibuat terenkapsulasi dari service lain. *Container* juga menspesifikasikan dengan jelas bagaimana proses *deployment* yang harus dilakukan dalam sebuah mesin, sehingga ketika sebuah aplikasi hendak dijalankan dalam mesin yang berbeda, *user* tidak perlu tahu proses apa saja yang harus dilakukan [9]. Contoh dari *container* adalah Docker, namun Docker tidak bisa melakukan *deployment* dalam banyak mesin. Maka dari itu Google mengembangkan *tools* yang bernama Kubernetes, Kubernet memungkinkan agar Docker bisa dalam satu saat bersamaan dijalankan dalam banyak mesin sekaligus. [9].

Figure 2‑5. Using Container to Deployment.

1. ***Serverless deployment.***

Ide dari *serverless deployment* adalah mengurangi interaksi dari pengguna dengan server. Segala hal yang berhubungan dengan infrastruktur server disembunyikan dari pengguna. Pengguna hanya dikenakan biaya penyewaan server saja, namun tidak perlu lagi melakukan pengaturan apapun pada server. Untuk melakukan *deployment*, pengguna meembuat package dari kode (misalnya ZIP file), lalu melakukan *upload* kepada penyedia jasa server dan melakukan pengaturan performa. Ada beberapa penyedia jasa lingkungan *serverless,* misalnya AWS Lambda, Google Cloud Function, Microsoft Azure. [6]

Kelebihan dari penggunaan *serverless* ini antara lain:

* Tidak perlu membuang-buang waktu untuk mengurus menejemen infrastruktur low-level. Pengguna bisa lebih fokus untuk mengembangkan aplikasinya saja.
* Arsitektur dari *serverless* sangat elastis. Server secara otomatis menghitung beban dari service yang digunakan agar tidak ada *resource* yang terbuang.

Kekurangan dari *serverless* antara lain:

* Adanya batasan lingkungan, misalnya server hanya bisa *support* untuk beberapa bahasa saja.

## 2.2.4 Metode Berkomunikasi Antar Service

Dengan memiliki modul service yang berbeda-beda, timbul sebuah masalah yang berkaitan dengan pertukaran informasi yang berasal dari banyak service. *Remote Procedure Invocation* (RPI)adalah protocol yang menyediakan teknik komunikasi antara service yang berbeda lokasi. Teknologi RPI ini ada yang menggunakan kode biner sebagai format pertukaran data, ada pula yang menggunakan format pesan XML seperti SOAP. Implementasi RPI ini berguna untuk mendapatkan data dengan sangat cepat yang dikirimkan melalui jaringan, hal yang menjadi keuntungan utama dari RPI adalah kemudahan penggunaannya. Contoh RPI lain yang menjadi fokus disini adalah Representational State Transfer (REST), point berikutnya akan menjelaskan mengapa REST menjadi pilihan terbaik untuk menangani proses komunikasi di microservice. [9]

**Representational State Transfer (REST).** REST adalah standar arsitektur web yang 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) [9]. Ide dari HATEOAS adalah *client* berhubungan dengan server hanya dengan menggunakan *link* yang telah disediakan.

Figure 2‑6. 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 Strategi Pengujian

Strategi pengujian berguna untuk memberikan gambaran dari test yang akan dilakukan terhadap *software*. Testing ini berguna untuk memberi tahu kepada proyek menejer, tester, dan tim pengembang apabila ditemukannya masalah dalam *software*. Strategi pengujian ini termasuk tujuan dari test, metode yang digunakan, sumber daya yang digunakan, juga lingkungan ketika menjalankan proyek. Test strategi mendeskripsikan seberapa tinggi resiko kesalahan (kegagalan) yang dapat terjadi [12].

Strategi yang dirancang dalam test kasus penulis adalah dengan pengujian 2 tahap, dimana tahap pertama adalah melakukan test untuk memastikan bahwa arsitektur microservice yang dibangun telah bekerja dengan baik dan telah memenuhi bisnis *requirement* seperti pada arsitektur sebelumnya. Sedangkan pada pengujian kedua, penulis akan melakukan uji perbandingan performa dengan arsitektur monolitik yang digunakan pada aplikasi saat ini.

## 2.3.1 Pengujian Arsitektur Microservice

Uji kelayakan arsitektur microservice menurut Martin Fowler adalah dengan melakukan 5 test yang mirip dengan software testing pada umumnya, test tersebut antara lain adalah:

1. *Unit Testing*. Bagian terkecil dalam software yang ditest untuk menentukan apakah unit tersebut bekerja sesuai harapan. Umumnya, unit yang dimaksud adalah kelas penyusun atau grup kecil yang menghubungkan kelas-kelas tersebut dan *method* yang terdapat dalam kelas.
2. *Integration Testing.* Goal dari test ini adalah membuktikan bahwa tidak terdapat masalah dari komunikasi dan interaksi dari setiap unit.
3. *Component Testing.* Komponen membatasi lingkup sebuah *software*. Dalam 1 buah komponen, dapat terdiri dari beberapa integrasi yang terjadi antara unit kelas.
4. *Contract testing.* Test yang memverifikasi bahwa pihak luar yang mengakses sebuah service akan mendapatkan hasil yang sesuai dengan harapan.
5. *End-to-end Testing.* Memverifikasi bahwa sistem memenuhi telah berhasil memenuhi kebutuhan dan sesuai dengan rancangan goal. Test dilakukan dari awal sampai dengan *output* yang dihasilkan.

Tujuan utama dari testing yang pertama adalah memastikan bahwa fungsi bisnis dari arsitektur yang baru sudah sama baiknya dengan arsitektur monolitik sebelumnya.

## 2.3.2 Uji Perbandingan Performa

Test kedua adalah menunjukan bahwa performa dari arsitektur microservice akan memberikan hasil yang lebih baik dari monolitk. Strategi pengujian yang akan dilakukan adalah dengan membuat point-point pembanding yang akan diuji, kemudian dari point tersebut akan dibuat *test-plan* masing-masing ang terdiri dari sekitar 1-3 buah *test-plan.* *Test result* kemudian akan disimpan dalam dokumen berupa *matrix traceability*.

Menurut Paulo Merson (Software Architecture di TCU; SOA/microservice trainer dan *consultan*), point pembanding tersebut adalah [12]:

1. *Deployability.* Lebih agile untuk untuk meluncurkan versi terbaru karena siklus *build, test, build* yang lebih pendek. Tingkat fleksibilitas yang lebih tinggi untuk menggunakan layanan keamanan, replikasi, persistensi, dan konfigurasi pemantauan.
2. *Reliability.* Kesalahan dalam microservice hanya akan berpengaruh pada microservice itu sendiri dan konsumennya, sedangkan dalam model monolitik kesalahan layanan dapat merusak seluruh sistem monolitik.
3. *Availability.* Waktu *downtime* yang dibutuhkan ketika ingin mengeluarkan versi terbaru dari microservice lebih sedikit dibandingkan monolitik.
4. *Scalability.* Setiap microservice dapat diskalakan secara terpisah menggunakan pool, cluster, grid. Karakter menyebar membuat microservice lebih fleksibel dibandingkan dengan monolitik.
5. *Modifiability and Management.* Sifat fleksibel untuk menggunakan framework, *libraries*, dan sumber data baru karena ditunjang sifat *loose-coupled* yang dimiliki microservice, modular komponen hanya dapat diakses oleh *contracts* (pihak yang berhak), dan sifat yang cenderung tidak mudah berubah menjadi *software* besar yang sulit ditangani. Upaya pengembangan aplikasi juga dapat dibagi dalam tim yang lebih kecil dan bekerja lebih mandiri.

Tujuan dari test performa ini adalah menunjukan bahwa arsitektur microservice yang baru akan lebih unggul dalam 5 point diatas apabila dibandingkan dengan monolitik.

Strategi berikutnya adalah membuat *test-plan* yang sesuai dengan ke-5 point tersebut, ­*test-plan* akan didokumentasikan dalam bentuk *matrix traceability* (Bab 4).

# BAB III ANALISA DAN PERANCANGAN

Pada bab ini, pertama-tama penulis akan membahas tentang proses bisnis dan arsitektur dari rumah sakit Apertura. Lalu akan membahas analisa terkait proses bisnis dan arsitektur yang baru untuk diterapkan pada aplikasi, termasuk segala komponen penyusun sistem yang baru mulai dari database, *tools*, IDE (*Integrated Development Environment*), framework yang digunakan dalam membangun aplikasi. Selanjutnya akan dibahas mengenai perancangan *service* dengan menggunakan konsep REST dan pembuatan API dari *service* tersebut.

## 3.1 Arsitektur Monolitik pada Software Apertura

Menurut Sam Newman, terdapat 2 parameter yang membedakan arsitektur monolitik dengan microservice, pertama adalah tingkat *coupling* dan yang kedua adalah tingkat *cohesion* dar*i* aplikasi tersebut. Pernyataan ini kemudian didukung oleh Chris Richardson yang menjelaskan bahwa kedua parameter ini dapat ditinjau dari berbagai sisi yang membentuk aplikasi tersebut, antara lain: data menejemen, cara berkomunikasi, dan metode *deployment* sebuah aplikasi.

**Data manajemen.** Aplikasi Apertura menerapkan model tersentralisasi yang sangat besar. Kurang lebih terdapat 120 tabel yang terdapat dalam 1 buah database tunggal*.* Namun semakin banyak model shared database yang digunakan, maka semakin tinggi pula derajat *coupling* aplikasi tersebut, dan tingginya derajat *coupling* merupakan salah satu ciri dari monolitik.

**Cara berkomunikasi.** Modul dalam database Apertura dapat mengakses langsung tabel milik modul lain dengan melakukan *querry* terhadap tabel tersebut. Cara berkomunikasi seperti ini seperti ini menjadi ciri dari arsitektur monolitik.

**Bentuk *deployment.***Tabel-tabel dan database Apertura disimpan dalam 1 buah database server yang sama. Penempatan server terpusat ini dianggap kurang menunjang konsep *high availability* apabila terjadi masalah pada server. Aplikasi menjadi sangat tergantung dengan server tunggal tersebut dan menjadi ciri derajat *coupling* yang tinggi.

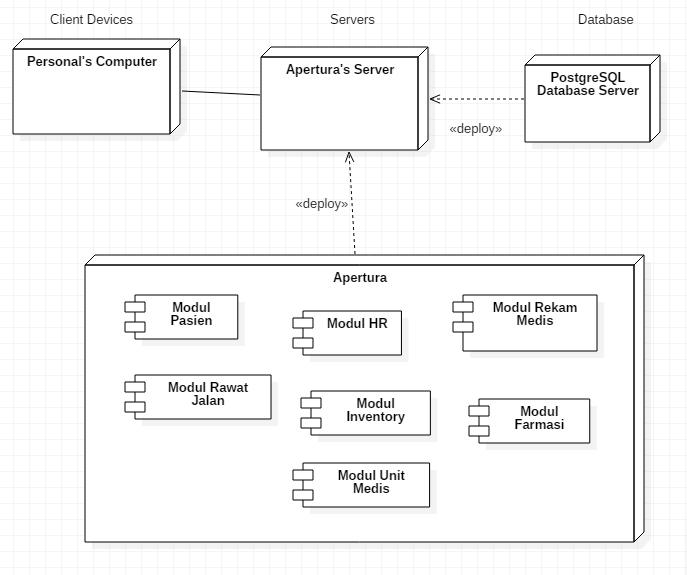
Arsitektur monolitik Apertura dapat digambarkan dengan deployment diagram dibawah:

Figure 3-1. Deployment Diagram

Berdasarkan analisis dari ketiga faktor diatas, maka dapat disimpulkan bahwa arsitektur dari aplikasi Apertura adalah monolitik.

## 3.2 Tinjauan Umum Proses Bisnis Pelayanan Rawat Jalan

Proses bisnis dalam rumah sakit Apertura melibatkan beberapa modul yang saling berinteraksi, modul-modul tersebut terdiri dari bagian yang lebih kecil lagi. Analisis berdasarkan proses bisnis berdasarkan kegunaannya akan lebih mudah untuk menentukan service yang akan terbentuk nantinya.

Modul-modul dari aplikasi Apertura antara lain: modul pasien, modul *HR (human resources)* yang meliputi data dokter, bidan, dan perawat, modul rekam medis, modul farmasi, modul inventori, modul akunting dan finansial, modul *invoicing* dan pembayaran, modul rawat jalan, modul rawat inap, modul pemeriksaan penunjang, dan modul integrasi SEP (Surat Eligibilitas Pasien) BPJS.

Dari semua modul diatas, penulis akan mengambil contoh kasus rawat jalan. Rawat jalan adalah tindakan perawatan pasien yang tidak menginap. Pasien yang datang akan mendaftar ke unit rawat jalan dan dicek apakah pasien tersebut terdaftar di BPJS, kemudian berdasarkan masalah pasien, pasien akan dirujuk ke unit medis yang ada di rumah sakit. Unit medis dari rumah sakit terdiri dari unit medis spesialis anak, spesialis jantung, dan spesialis penyakit dalam. Tiap unit medis memiliki satu atau lebih dokter spesialis dari bidang unit medis tersebut. Pasien kemudian akan ditangani oleh dokter yang bertugas di unit medis tersebut. Proses penanganan pasien dimulai dari konsultasi keluhan, pemeriksaan penunjang, dan diagnosis penyakit.

Hasil dari pemeriksaan tersebut akan disimpan dalam rekam medis pasien di rumah sakit. Modul rawat jalan akan mengeluarkan resep obat yang dapat pasien ambil di farmasi. Modul rawat jalan juga akan mengeluarkan detail faktur yang dibutuhkan untuk proses pembayaran.

Maka dari itu, modul rawat jalan akan melibatkan modul pasien, HR, unit medis, farmasi, rekam medis, pemeriksa penunjang, pembayaran dan penagihan, integrasi BPJS, dan modul rawat jalan itu sendiri.

Berikut adalah penggambaran proses bisnis dari pelayanan rawat jalan:



Figure 3-2. Proses Bisnis Rawat Jalan

## 3.3 Pembahasan Modul dan Kelas Penyusun

Pada bagian ini akan dijelaskan fungsi dan kelas-kelas dari setiap modul dalam kaitannya dengan proses bisnis di rumah sakit.

1. **Modul Pasien.** Modul pasien berfungsi untuk menyimpan, memperbaharui, pencarian, dan menampilkan data pasien. Data pasien juga meliputi detail lengkap dari data keluarga atau kerabat yang menjadi penanggung jawab pasien. Kelas dari modul pasien adalah pasien itu sendiri.
2. **Modul *Human Resource*.** *Human Resource* adalah modul yang mengelola semua pengguna dan pegawai dalam rumah sakit, modul ini dapat disebut juga sebagai modul karyawan. Beberapa contoh yang termasuk dalam modul ini adalah dokter, bidan, dan perawat. Modul ini menjadi modul dasar yang dibutuhkan modul lain, karena terkait pencatatan pengguna yang melakukan input data. Kelas-kelas dari modul ini meliputi dokter, bidan, dan perawat. Namun dalam contoh kasus rawat jalan yang akan diangkat, kelas yang diambil hanya dokter saja.
3. **Modul Unit Medis.** Unit medis dari rumah sakit terdiri dari unit medis spesialis anak, spesialis jantung, dan spesialis penyakit dalam. Tiap unit medis memiliki satu atau lebih dokter spesialis dari bidang unit medis tersebut. Pasien akan ditangani oleh dokter yang bertugas di unit medis tersebut.
4. **Modul Rekam Medis.** Rekam medis bertugas mencatat semua riwayat medis milik pasien yang meliputi hasil diagnosa, resep dan obat yang pernah diberikan, alergi, penyakit kronis, riwayat tindakan atau perawatan medis, dan hasil pemeriksaan penunjang. Pemeriksaan penunjang dapat berupa banyak aksi, tergantung apa yang dibutuhkan dokter. Pemeriksaan penunjang antara lain seperti radiologi, tes darah, tensi tekanan, dan lain lain. Fungsi dari pemeriksaan penunjang adalah menunjang ditegakannya diagnosis.
5. **Modul Farmasi.** Modul farmasi bertugas untuk mengatur dan menyimpan obat-obatan. Modul farmasi menerima permintaan obat dari bagian IGD, rawat inap, rawat jalan, dan juga penjualan umum. Dalam aplikasi Apertura, modul ini juga meliputi fungsi dari apotek. Untuk IGD, rawat inap, dan rawat jalan pasti akan memiliki resep, namun untuk penjualan obat umum dapat menggunakan resep rujukan maupun tidak.
6. **Modul Inventori.** Inventori dibagi menjadi 2 tipe, yaitu barang dan jasa. Barang meliputi obat, alat kesehatan (alkes), dan barang selain obat dan alkes. Adapun jasa yang dimaksud adalah jasa pelayanan kesehatan yang disediakan rumah sakit. Inventori berkaitan langsung dengan modul farmasi, karena modul farmasi membutuhkan data barang yang dijual.
7. **Modul Rawat Jalan.** Rawat jalan adalah pelayanan medis kepada pasien untuk tujuan perawatan tanpa mengharuskan pasien tersebut untuk menginap.Rawat jalan menjadi perantara interaksi dari pasien dengan unit medis, rawat jalan menyimpan data perawatan yang kemudian akan disimpan dalam rekam medis. Hasil dari rawat jalan akan kemudian dibutuhkan oleh bagian pembayaran dan juga farmasi.

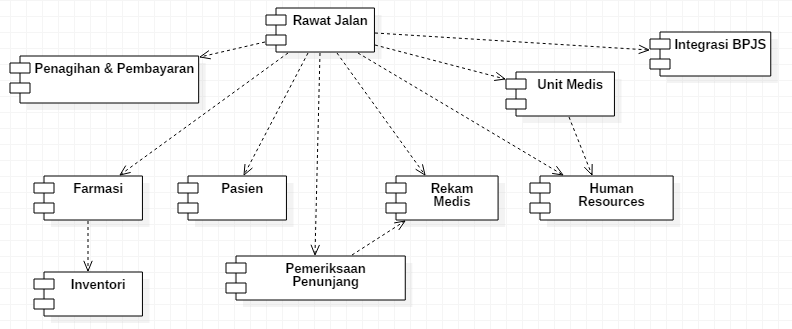
Berikut adalah penggambaran relasi dari modul rawat jalan menggunakan komponen diagram.

Figure 3‑3.Komponen diagram rawat jalan

## 3.4 Identifikasi Teknologi yang Digunakan pada Aplikasi Apertura

Saat ini teknologi aplikasi yang digunakan Rumah Sakit Apertura menggunakan arsitektur monolitik dengan database yang tersentralisasi pada 1 buah server. Database Apertura sendiri menggunakan PostgreSQL 9. PostgreSQL merupakan salah satu object-relational database management system (ORDMS) yang tersedia secara *open source*. Database Apertura terdiri dari kurang lebih 120 tabel yang saling berelasi. Aplikasi Apertura digunakan oleh beberapa kelompok pengguna, antara lain bagian kasir rumah sakit, registrasi rawat jalan, registrasi rawat inap, kasir apotik, kasir rawat jalan, dokter, staf administrasi umum, staf keuangan, staf rekam medis, staf administrasi BPJS.

Aplikasi Apertura diimplemetasikan untuk diakses dalam bentuk aplikasi *desktop. User interface* yang ditampilkan dalam aplikasi tidak terpisah secara moduler, melainkan satu kesatuan aplikasi besar yang kemudian dipisahkan berdasarkan kebutuhan *user* yang menggunakan aplikasi.

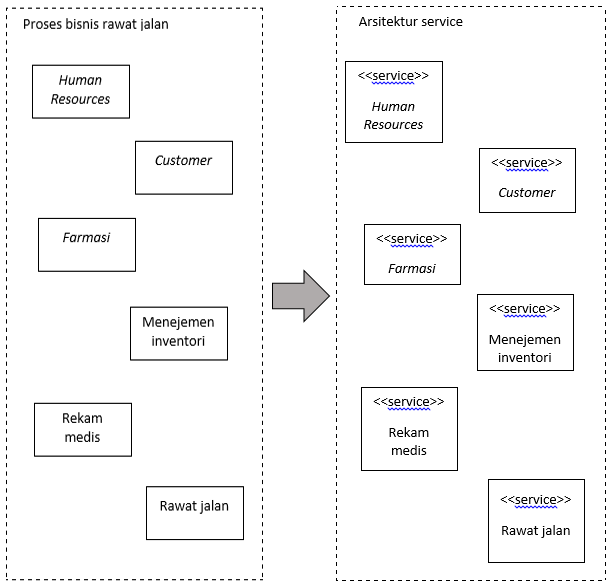
## 3.4.1 Kekurangan Arsitektur Monolitik

Kelemahan dari sistem Apertura sebenarnya berkaitan dengan kelemahan dari arsitektur monolitik itu sendiri. Berdasarkan analisis yang dilakukan, kelemahan dari arsitektur monolitik itu sendiri adalah:

* Tingkat *coupling* yang terlalu tinggi. Semua modul dalam sistem saling terikat, mengakibatkan kendala dalam proses pengembangan sistem. Perubahan dalam 1 buah objek harus diketahui oleh objek lain. Proses *deployment* akan terhambat karena apabila hendak melakukan tes terhadap 1 buah modul, maka keseluruhan aplikasi harus di jalankan, mengakibatkan beban hardware lebih besar dan memakan waktu lebih banyak.
* Penyimpanan data yang terpusat pada 1 buah tempat (database) kurang baik. Apabila database mengalami *down*, maka keseluruhan modul dari aplikasi tidak bisa melakukan pekerjaannya.
* Isu keamanan. Penyimpanan data dalam 1 buah database tidak cukup aman. Misalnya ada pihak yang tidak berwenang berhasil mengakses database, maka seluruh data aplikasi akan bocor.
* Sistem tidak cukup baik ketika menangani banyak user. Sistem akan kewalahan apabila banyak pengguna yang melakukan request dalam satu waktu, isu yang ditimbulkan adalah performa.

## 3.5 Perancangan Arsitektur Microservice

Dengan menggunakan pattern dekomposisi berdasarkan kemampuan bisnisnya, maka service yang akan terbentuk dapat dibayangkan berdasarkan modul bisnis itu sendiri. Dalam kasus rawat jalan yang diangkat, maka modul yang saling berelasi adalah modul *customer,* modul HR, modul inventori, modul farmasi, dan modul rawat jalan. Ke 5 modul ini akan dibuat menjadi service yang mandiri.



## 3.5.1 Pemilihan Model Penyimpanan Data

Setelah kelas-kelas service terbentuk, hal berikutnya yang harus diperhatikan adalah pemodelan menejemen data. Apabila ditinjau dari service yang dihasilkan, maka akan lebih baik menerapkan pattern database per service, dimana satu service memiliki sebuah set database yang hanya bisa diakses oleh service tersebut dan tidak bisa saling mengakses database service yang lain.

Pemodelan database juga bergantung dengan bagaimana model database itu sendiri. Apabila model database yang digunakan nanti adalah relational database, maka service tersebut akan memiliki sebuah set database yang terdiri dari beberapa schema berhubungan yang hanya bisa diakses oleh service tersebut. Apabila database yang digunakan adalah non-relational database, maka cukup sebuah schema tabel untuk 1 service.

## 3.5.2 Pemilihan Server Basisdata

Aplikasi akan menggunakan 2 buah database untuk kegunaan yang berbeda. Database server yang pertama akan tetap sama seperti yang saat ini sistem gunakan yaitu PostgreSQL versi 9.4. Namun untuk kasus seperti rekam medis yang berbentuk multimedia (video, audio, foto) peneliti akan menggunakan MongoDB yang dinilai mudah, cepat, dan *open source*. Walaupun database yang digunakan sama dengan yang digunakan oleh sistem sebelumnya, namun perancangan dalam database sangat berbeda terlebih setelah memilih model penyimpanan data.

## 3.5.3 Pemilihan Komunikasi Antar Service

Setelah membuat service dan database yang digunakan, hal berikutnya adalah membuat API dari service tersebut agar dapat diakses menggunakan *web service*. Berdasarkan kebutuhan arsitektur microservice sendiri, model *web service* yang akan digunakan adalah RESTful (Representational State Transfer) dengan format data yang dikirim berupa JSON (JavaScript Object Notation). Server sementara yang akan digunakan selama tahap perancangan adalah mesin pribadi milik peneliti. Setelah perancangan selesai dan siap, server baru akan dipindahkan ke server milik rumah sakit. Untuk tes pemanggilan API sementara dapat dilakukan dalam satu jaringan yang sama.

## 3.5.4 Rancangan Deployment

Model *deployment* yang akan digunakan adalah *multiple service per host,* dikarenakan lebih cocok dalam kasus penelitian yang hanya mengambill proses rawat jalan. Semua set service yang terbentuk akan di *upload* dalam beberapa server agar dapat membuktikan tercapainya *high availability*. Apertura sendiri memiliki 2 buah server aktif yang dapat digunakan ketika proses *deployment*, serta server lainnya dapat menggunakan mesin pribadi untuk melakukan pemanggilan service dari server lain.

## 3.6 Batasan Implementasi

Adapun batasan implementasi dalam pengembangan aplikasi yaitu:

1. Rancangan aplikasi tidak dapat menggunakan layanan integrasi BPJS karena harus adanya akun terverifikasi.
2. Server *cloud* yang dapat digunakan sebanyak 2 buah milik Apertura. Alasan tidak menambah server dikarenakan server privat harus berbayar.
3. Dikarenakan *deployment*  service ada yang dilakukan di server local (mesin pribadi), maka pemanggilan API dibatasi dalam 1 jaringan yang sama.

## 3.7 Lingkungan Implementasi

Adapun spesifikasi mesin pribadi ketika melakukan proses implementasi adalah sebagai berikut:

* Perangkat keras: 1 buah komputer sebagai server, 1 buah komputer sebagai *client*.
* Spesifikasi komputer: 64-bit prosesor dan sistem operasi.
* OS: Windows 10 Pro.
* Processor: Intel(R) Core i3-4150 CPU, 3.50GHz
* Memory: 8 GB RAM.
* Storage: 50 MB.
* *Integrated Development Environment* (IDE): Netbeans / Eclipse neon 3.
* Java version: 1.8.0\_60
* DB : PostgreSQL 9.6

[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

[11] Ry, K. Siva Prasad (2017). Beginning Spring Boot 2 Applications and Microservices with the Spring Framework. Springer Science and Business Media \_ Apress

[12] Ammann, Paul and Offutt, Jeff (2008). Introduction to software testing. New York: Cambridge University Press.