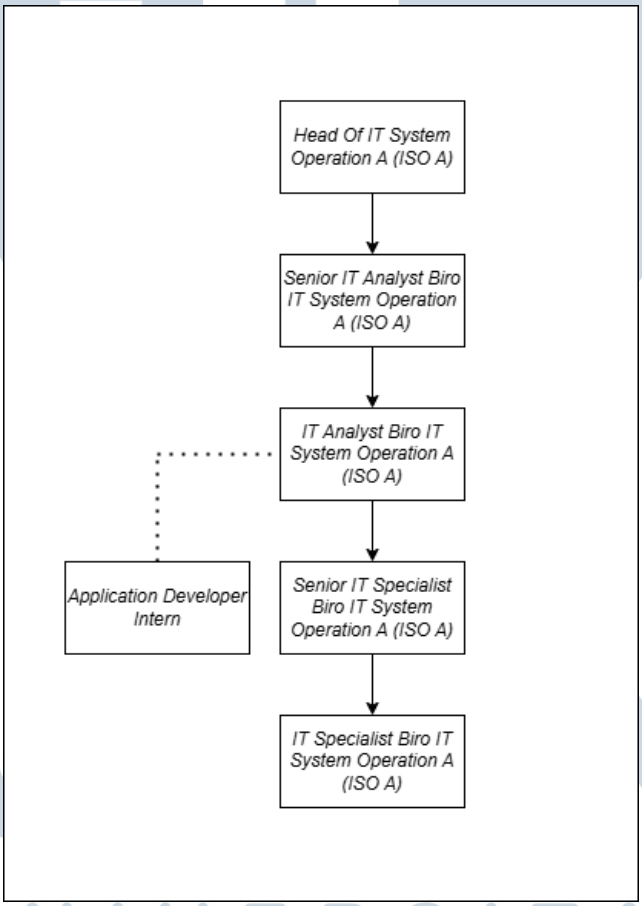
BAB 3
PELAKSANAAN KERJA MAGANG

3.1 Kedudukan dan Koordinasi

Kegiatan Kerja magang dilaksanakan di biro ISO A, yang berada di bawah subgrup *IT System Operation*(ISO), dengan posisi sebagai *Application Developer Intern*.



Gambar 3.1. Kedudukan Intern di Biro ISO A pada PT Bank Central Asia Tbk

Sumber : [9]

Berdasarkan Gambar 3.1, posisi *Application Developer Intern* merupakan bagian dari tim teknis yang berperan sebagai pelaksana pendukung dalam kegiatan pengembangan sistem *backend*. *Intern* tidak termasuk dalam jalur struktural utama, namun terhubung langsung dengan *IT Analyst*, yang bertugas sebagai mentor teknis dan pembimbing harian. Hubungan ini digambarkan dengan garis putus-putus dalam bagan, yang menunjukkan peran pembinaan non-struktural. Peserta magang

memperoleh tugas dan supervisi dari Galuh Dhian Wibowo selaku *IT Analyst* dalam menjalankan aktivitas magang sehari-hari.

IT Analyst memberikan tugas-tugas pengembangan, melakukan *code review*, dan memberikan masukan terhadap hasil kerja peserta magang. Selain itu, peserta magang juga memperoleh pengarahan strategis dari Suryanto Chandra selaku *Senior IT Analyst* dan pemantauan struktural dari *Head of IT System Operation A*. Dengan demikian, peserta magang tidak bekerja secara mandiri, melainkan menjadi bagian dari sistem kerja terstruktur yang terintegrasi dengan anggota tim lainnya.

Dalam pelaksanaan tugas sehari-hari, peserta magang berkoordinasi dengan berbagai staf *IT System Operation A* yang tersebar dibeberapa wilayah kerja PT Bank Central Asia. Koordinasi ini dilakukan melalui beberapa metode yang fleksibel dan efisien. Salah satu media utama yang digunakan adalah *Microsoft Teams*, yang memungkinkan komunikasi secara *real-time* dalam bentuk *chat, voice call*, maupun *video conference*. *Platform* ini juga digunakan untuk keperluan rapat tim, *update* status proyek, dan konsultasi teknis dengan mentor atau anggota tim lainnya.

Untuk pengelolaan kode program, tim Biro *IT System Operation A* menggunakan Gitlab sebagai *platform* utama *version control*. Peserta magang diajarkan untuk membuat *branch*, melakukan *commit*, *push*, serta *merge request* melalui Gitlab sebagai bagian dari praktik *DevOps*. Seluruh pengembangan aplikasi dilakukan secara kolaboratif di dalam *repository* yang telah diatur oleh tim. Setiap perubahan kode yang diajukan oleh intern akan melalui proses *review* oleh mentor sebelum diintegrasikan ke dalam *branch* utama. Dengan menggunakan GitLab, transparansi dan kontrol terhadap versi aplikasi dapat terjaga dengan baik.

3.2 Tugas yang Dilakukan

Selama pelaksanaan magang di PT Bank Central Asia Tbk., terdapat beberapa tugas yang dilakukan dengan berfokus pada Implementasi layanan *backend EMS-Service* menggunakan *Spring Boot*, sebagai bagian dari ekosistem *microservice* internal perusahaan. Tugas utama yang dilakukan melibatkan pemrograman, dokumentasi teknis, serta kolaborasi dengan tim *development* dalam rangka mendukung pengembangan sistem *Event Monitoring System (EMS)* yang dimiliki BCA.

Adapun tugas-tugas yang dilakukan antara lain:

1. Mempelajari dan Menganalisis Arsitektur Sistem yang Ada

Peserta magang ditugaskan untuk memahami alur sistem yang telah berjalan, termasuk bagaimana sistem tersebut dikembangkan.

2. Melakukan Pengembangan Layanan *Backend* Menggunakan Spring Boot
   Peserta magang terlibat dalam proses pembuatan dan pengembangan *EMS-Service* menggunakan *framework* Spring Boot, termasuk konfigurasi project, pembuatan *controller*, *service*, dan *repository* sesuai kebutuhan bisnis.

3. Pembuatan dan Dokumentasi Endpoint
   Peserta magang mengimplementasikan sejumlah *Restful API* untuk menerima, mencatat, dan mengakses data kejadian sistem. Seluruh *endpoint* terdokumentasi menggunakan *Swagger* agar dapat dipahami dan digunakan oleh tim *internal* maupun layanan lain dalam ekosistem BCA.

4. Penerapan Prinsip *Layered Architecture* dan *Separation of Concerns*
   Dalam proses *development*, Peserta magang menerapkan prinsip *Separation of Concerns,* memastikan adanya pemisahan tanggung jawab melalui arsitektur *layered* (*Controller, service, repository, integration).* Selain itu, Peserta magang memastikan bahwa layanan *EMS-Service* dapat berintegrasi dengan baik dalam ekosistem *microservices* BCA melalui komunikasi HTTP berbasis *API*

5. Kolaborasi dengan Tim dan *Daily Meeting*
   Peserta magang mengikuti *Daily Meeting* untuk menyampaikan *progress*, hambatan, dan koordinasi tugas harian

6. Dokumentasi Teknis dan Deployment Support
   Peserta magang berkontribusi dalam penyusunan dokumentasi teknis yang berkaitan dengan proses pengembangan aplikasi. Selain itu, peserta magang juga membantu dalam proses instalasi *requirement* yang diperlukan untuk menjalankan aplikasi, khususnya sebelum dilakukan *deployment* ke lingkungan *staging* atau *testing*. Kegiatan ini dilakukan bekerja sama dengan tim *DevOps*, terutama saat proses implementasi aplikasi ke *cloud platform OpenShift*, yang digunakan sebagai lingkungan *hosting* internal perusahaan.

3.3 Uraian Pelaksanaan Magang

Pelaksanaan kerja magang diuraikan seperti pada Tabel 3.1.

Tabel 3.1. Pekerjaan yang dilakukan tiap minggu selama pelaksanaan kerja magang

Minggu Ke -	Pekerjaan yang dilakukan
1	Pengenalan lingkungan kerja pada Biro ISO A PT Bank Central Asia Tbk, melakukan registrasi akun serta hal-hal yang dibutuhkan untuk memulai magang.
2	Mempelajari materi terkait *tools* dan teknologi yang akan digunakan selama magang. Setelah itu, membuat sebuah *project* sederhana untuk mengimplementasikan Springboot yang telah dipelajari.
3	Membuat sebuah *RestAPI* untuk aplikasi telegram dengan Springboot, serta Mempelajari dan Implementasikan Webhook dan Springboot dalam project tersebut
4	Melakukan *debugging* dan perbaikan *logic* bisnis pada *service DCNotif*, termasuk analisis *error* dari *log*, penyesuaian *timezone* dan format waktu, serta pengujian dan dokumentasi hasil perubahan.
5	Mempelajari alur kerja dan struktur proyek *Dispatcher Task* (DITA), termasuk sistem *login* berbasis *role* dan *token*. Melakukan *debugging*, penyesuaian format waktu GMT+7 pada *service DCNotif* dan *Job Time*, serta *testing*, *simulasi error handling*, penyesuaian *interval schedule*, dan instalasi sertifikat *API*.
6	Fokus pada implementasi fitur *pagination* di beberapa *controller* proyek *DCNotif*, seperti *SearchController, ReportController*, dan *IncidentController*, termasuk *testing*, perbaikan *error*, dan integrasi dengan *response schema*. Selain itu, mempelajari dan mengimplementasikan *Swagger* untuk dokumentasi *API* pada proyek *Spring Boot*.
7	Mulai Mengerjakan proyek baru bernama EMS-Service sebagai bagian dari API Gateway untuk meneruskan data kejadian dari EMS ke APICenter.

Minggu Ke -	Pekerjaan yang dilakukan
8	Melanjutkan pengembangan proyek EMS, dengan fokus pada debugging dan perbaikan alur logika pemrosesan data. Membangun logika untuk pencarian data berdasarkan parameter tertentu dan pengambilan waktu (*GetTime*) dari tabel lain sebagai referensi.
9	Melakukan pembangunan ulang proyek EMS karena adanya perubahan alur proses.
10	Melanjutkan pengembangan proyek EMS, dengan fokus pada pembuatan *Logic* Bisnis, Validasi *Input*, dan Exception
11	Melakukan Testing pada project EMS dan implementasi dokumentasi dengan Swagger

Pada minggu pertama pelaksanaan magang, kegiatan diawali dengan pengenalan terhadap lingkungan kerja serta para karyawan yang berada di biro ISO-A. Pegawai di biro ini bekerja di beberapa lokasi berbeda, yaitu Jakarta, Surabaya, dan Jogjakarta, Sehingga koordinasi dan pertemuan umumnya dilakukan secara daring melalui Microsoft Teams. Selama masa magang, perusahaan menyediakan berbagai *tools* pendukung yang dapat digunakan oleh peserta magang, namun diperlukan proses registrasi dan pengajuan akses terlebih dahulu. Selain itu, peserta magang juga diberikan sejumlah materi pembekalan seperti pentingnya menjaga data pribadi perusahaan, etika sebagai seorang bankir, dan materi pendukung lainnya sebelum mendapatkan akses ke akun serta perangkat yang akan digunakan.

Pada minggu kedua, kegiatan dimulai dengan mempelajari berbagai *tools* dan teknologi yang akan digunakan selama masa magang, khususnya terkait konsep *Microservices* dan *API Gateway*. Selain itu, dilakukan instalasi beberapa aplikasi pendukung seperti IntelliJ IDEA, Postman, Maven, Java, dan PgAdmin untuk menunjang proses pengembangan sistem *backend* berbasis Spring Boot. Sebagai bentuk implementasi dari materi yang telah dipelajari, dilakukan pembuatan aplikasi sederhana dengan arsitektur *microservice* menggunakan Spring Boot dan PostgreSQL sebagai *database*. Selanjutnya, dilakukan pula pembelajaran mengenai struktur folder dalam proyek Spring Boot serta pembuatan *Error Schema* dan *Output Schema* yang digunakan dalam proses pengujian melalui Postman.

Pada minggu ketiga, kegiatan difokuskan pada pengembangan RestAPI untuk *chatbot* pada aplikasi Telegram menggunakan Spring Boot, serta mempelajari dan mengimplementasikan konsep Webhook dalam proyek tersebut. Dan

pemahaman mengenai cara kerja API Telegram melalui Webhook diperdalam untuk memastikan komunikasi antara sistem dan aplikasi telegram dapat berjalan dengan baik. Selain itu dilakukan pengujian endpoint dengan postman guna mengirim data dalam format JSON. Kegiatan ini juga mencakup eksplorasi awal terhadap penggunaan Springboot Kotlin dan Node.js sebagai referensi untuk pengembangan proyek lanjutan.

Pada minggu keempat, fokus utama kegiatan adalah melakukan debugging dan perbaikan *logic* bisnis pada *service* DCNotif dalam proyek berbasis *microservices*. Kegiatan diwali dengan *daily meeting* bersama tim divisi untuk menyampaikan *progres* dan rencana kerja harian. Perbaikan dilakukan terhadap *logic* pengambilan waktu khususnya pada bagian *StartTime* dan *ToTime*, agar sesuai dengan kebutuhan sistem. Proses ini melibatkan *review code*, pengujian layanan, serta diskusi bersama mentor untuk memperoleh *feedback* atas perubahan yang dilakukan. Selain itu, dilakukan penelusuran terhadap *error* yang muncul dengan memanfaatkan *log* dari Intellij Idea dan pengujian menggunakan Postman. Isu utama yang dianalisis adalah terkait timezone dan format waktu GMT, yang berpotensi menyebabkan ketidaksesuaian pada data.

Pada minggu kelima, kegiatan berfokus pada pemahaman alur kerja proyek *Dispatcher Task* (DITA), termasuk sistem *login* berbasis token dan *role user*. Analisis dilakukan terhadap *source code* dan *logic login* untuk memahami cara kerja autentikasi dan pengelolaan *session* pengguna. Selain itu, dipelajari pula penggunaan *framework* Next.js yang digunakan dalam pengembangan *frontend* proyek DITA. Di sisi *backend*, dilakukan *debugging* dan penyesuaian terhadap format waktu GMT+7 pada *service* DCNotif dan *job time* untuk memastikan kesesuaian waktu dengan zona lokal. Proses ini dilanjutkan dengan simulasi *error handling* pada penjadwalan *job* serta analisis penyebab error yang berkaitan dengan konsersi waktu. *Certificate* API dari website eksternal juga diinstal dan dikonfigurasi ke dalam environment Java lokal untuk kebutuhan autentikasi API. Selain itu, dilakukan penyesuaian interval waktu *schedule* dari 3 menit menjadi 2 menit untuk meningkatkan responsivitas sistem.

Pada minggu keenam, fokus kegiatan diarahkan pada pengembangan dan penyempurnaan fitur *pagination* dalam proyek DCNotif. Proses ini mencakup implementasi *pagination* pada beberapa controller seperti SearchController, ReportController, dan IncidentController, serta pengujian fungsionalitas *pagination* di berbagai *endpoint*, termasuk data laporan hari ini dan tujuh hari terahkir. Selama proses pengujian, dilakukan evaluasi menyeluruh terhadap potensi *error*

yang mungkin muncul dari perubahan logika *pagination*. Selain itu, dilakukan perbaikan terhadap *format output response*, dengan menyesuaikan struktur *respons* agar terbungkus dalam *format page*. Ditengah kegiatan tersebut, peserta magang juga mempelajari dan mengimplementasikan swagger sebagai alat dokumentasi API untuk *project spring Boot*. Sebagai langkah awal, dibuat sebuah proyek sederhana untuk integrasi Swagger, yang kemudian diadaptasi dan diterapkan ke dalam proyek utama DCNotif. Setelah implementasi, dokumentasi API dengan Swagger ditambahkan pada beberapa *controller* untuk memudahkan pemahaman tim terhadap struktur dan fungsionalitas endpoint. Progress pekerjaan terus dipantau melalui meeting harian bersama mentor serta tim satu divisi. Seluruh perubahan yang telah dilakukan, termasuk perbaikan dan penambahan fitur, kemudian di push ke *repository* GitLab BCA, dilanjutkan dengan pembuatan *merge request* sebagai bagian dari alur kolaborasi pengembangan.

Pada minggu ketujuh, Kegiatan magang difokuskan pada pengembangan proyek baru bernama EMS-Service, yang berperan sebagai *API Gateway* dalam ekosistem *microservices*. Langkah awal dimulai dengan *briefing* bersama mentor untuk memahami tujuan dan ruang lingkup proyek, dilanjutkan dengan proses *pull project* dari GitLab BCA dan mempelajari struktur file dan alur kerja proyek EMS-Service. Setelah memahami struktur proyek, dilakukan pengembangan awal berupa penyimpanan data transaksi ke *database* Transaksi, serta proses pencocokan data yang kemudian diteruskan dan disimpan ke dalam *database Master*. Selama proses ini, peserta magang juga menerapkan format waktu dengan pola tertentu pada setiap *service* guna memastikan konsistensi waktu antar database. Proses integrasi antar database kemudian dilanjutkan dengan pembuatan *service* MasterEMS, termasuk skema penyimpanan data dari Transaction EMS ke data master an penyesuaian *response schema* yang digunakan saat data berhasil disimpan. Dalam pengembangannya, peserta magang aktif berdiskusi dengan tim untuk mengidentifikasi kendala dan mencari solusi teknis. Fitur penting lain yang dikembangkan adalah pencatatan status pengiriman berupa *statusSending* dan *timeSending* jika data berhasil diteruskan ke *service* eksternal, serta pembuatan *log error* untuk mencatat jika terjadi kegagalan pengiriman data. Terahkir, dilakukan penyesuaian agar *response schema* EMS-Service dapat menyesuaikan format balikan dari Service Center secara dinamis, sehingga memudahkan pemantauan alur data dan *debugging*.

Pada minggu kedelapan, difokuskan pada pengembangan lanjutan proyek EMS-Service, khususnya dalam aspek *debugging* dan penyempurnaan alur logika

pemrosesan data. Proses dimulai dengan identifikasi penyebab error yang terjadi pada alur integrasi data, kemudian dilakukan *debugging* secara menyeluruh untuk memastikan validitas dan konsistensi data antar service. Salah satu fokus utama adalah membangun logika pencarian data berdasarkan parameter tertentu, sehingga sistem dapat mengambil data yang lebih relevan dan efisien sesuai kebutuhan pengguna. Selain itu, peserta magang juga mengembangkan logika pengambilan waktu (*GetTime*) dari tabel referensi lain yang berkaitan, yang sangat penting untuk memastikan ketepatan waktu dalam proses pencatatan status dan histori transaksi.Aktivitas teknis ini diiringi dengan berbagai diskusi dan kolaborasi, baik melalui *meeting* rutin bersama divisi lain maupun koordinasi intensif dengan mentor dan tim satu divisi untuk membahas kendala teknis serta solusi implementatif. Tujuan dari diskusi ini adalah untuk menyelaraskan pengembangan sistem agar sejalan dengan standar internal perusahaan dan kebutuhan sistem secara keseluruhan. Meskipun proyek sudah memasuki tahap lanjut, proses *debugging* tetap menjadi bagian penting agar sistem dapat berjalan stabil dan responsif. Secara keseluruhan, minggu ini menjadi momentum untuk memperkuat pondasi sistem EMS-Service melalui perbaikan logika dan validasi waktu, serta meningkatkan kemampuan *debugging* dan komunikasi dalam tim pengembangan.

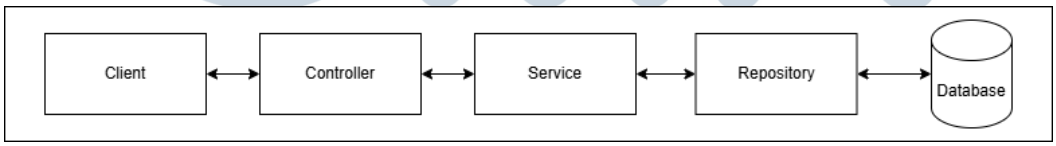
Pada minggu kesembilan, yaitu melakukan pembangunan ulang terhadap proyek EMS-Service akibat perubahan alur bisnis yang signifikan. Sebelumnya, data event dikirim dan disimpan terlebih dahulu ke tabel transaksi, namun kini diarahkan untuk masuk ke tabel master terlebih dahulu sebelum diproses lebih lanjut. Perubahan ini menginginkan penyesuaian besar pada struktur alur data dan logika service yang telah dibangun sebelumnya. Proses diawali dengan analisis ulang terhadap *requirement* baru serta melakukan diskusi bersama mentor dan tim untuk menyusun ulang flow yang sesuai. Setelah itu, dilakukan pembaruan struktur *class* dan *entity* yang terkait dengan penyimpanan awal ke MasterEMS. Logika validasi, penyimpanan, dan pengecekan duplikasi data pada MasterEMS disempurnakan agar lebih akurat. Kemudian, dibuat penghubung untuk mengalirkan data dari Master ke Transaction hanya jika status tertentu terpenuhi. *Debugging* dilakukan secara menyeluruh untuk memastikan bahwa perubahan tidak menimbulkan *error* baru. Selain itu, dilakukan testing dengan berbagai skenario untuk memvalidasi keakuratan data dan alur penyimpanan. Seluruh hasil perubahan didokumentasikan sebagai acuan pengembangan selanjutnya. Dengan adanya revisi alur ini, sistem menjadi lebih terstruktur dan siap untuk dikembangkan ke tahap integrasi berikutnya.

Pada minggu kesepuluh, peserta magang melanjutkan pengembangan proyek *EMS-Service* dengan fokus pada penyempunaan logika bisnis, validasi *input*, dan penanganan *exception*. Pada minggu ini, dilakukan perbaikan terhadap beberapa logika bisnis yang sebelumnya belum optimal. Selain itu, ditambahkan validasi pada *request* data agar field penting tidak bernilai *null* atau kosong. Khusus untuk field *condition*, sistem dibatasi hanya menerima dua nilai *valid*, yaitu normal dan alarm. Apabila *input* tidak sesuai, maka sistem akan menolak permintaan dan mengembalikan pesan *error* melalui mekanisme *exception handling*.

Pada minggu kesebelas, melakukan pengujian (*testing*) terhadap project EMS-Service dengan berbagai skenario, antara lain pengujian validasi input, pengiriman data EMS ke APICenter, serta penanganan error saat terjadi kondisi gagal koneksi. Tujuan pengujian ini adalah untuk memastikan sistem telah berjalan sesuai logika bisnis yang diimplementasikan dan memberikan response yang konsisten. Selain itu, juga dilakukan implementasi dokumentasi API menggunakan Swagger, sehingga setiap *endpoint* yang tersedia dapat terdokumentasi secara otomatis dan dapat diakses melalui antarmuka Swagger UI untuk keperluan testing maupun integrasi antar tim.

3.3.1 *Architecture* dan *Framework* yang Digunakan

Proyek ini dibangun menggunakan *framework Spring Boot*. *Spring Boot* dipilih karena kemampuannya untuk cepat dijalankan, dan mudah dikonfigurasi. Dalam konteks *microservices*, *Spring Boot* memungkinkan setiap layanan dikembangkan secara terpisah namun tetap terintegrasi melalui komunikasi antarlayanan seperti *REST API*. *Framework* ini dimanfaatkan untuk membangun sisi *backend* dari sistem ISO A.



Gambar 3.2. Arsitektur *Layered* Aplikasi *Back-end*
Sumber : [10]

Aplikasi yang dikembangkan menerapkan arsitektur *layered*, yang alur kerjanya ditunjukkan pada Gambar 3.2:

1. *Client*: *Client* Merupakan sistem lain yang mengirimkan permintaan (*request*) dan menerima tanggapan (*response*) dari sistem melalui *controller*.

Client dapat berupa aplikasi *frontend*, layanan pihak ketiga, atau sistem lain yang berkomunikasi dengan *back-end* menggunakan protokol HTTP atau Restful API.

2. *Controller*: *Controller* berperan sebagai titik masuk utama (*entry point*) dari setiap permintaan yang dikirim oleh *client*. Komponen ini bertugas menangani *request*, memvalidasi data (jika diperlukan), dan kemudian meneruskannya ke *layer service* yang sesuai untuk diproses lebih lanjut.

3. *Service: Layer service* memuat logika bisnis utama dari aplikasi. Di sinilah seluruh proses pengolahan data dilakukan, termasuk perhitungan, validasi, dan pengambilan keputusan. *Service* dapat berinteraksi dengan *repository* atau memanggil *service* lain dalam arsitektur yang lebih kompleks.

4. *Repository*: *Repository* merupakan *layer* yang bertanggung jawab terhadap akses data. Ia mengelola komunikasi dengan *database* menggunakan operasi CRUD (*Create, Read, Update, Delete*). Dalam aplikasi ini, interaksi ke *database* dilakukan menggunakan *JDBC Template*, yang memungkinkan eksekusi perintah SQL secara langsung dan terkontrol.

5. *Database*: *Database* adalah komponen penyimpanan utama yang menyimpan seluruh data yang digunakan oleh aplikasi. Data yang tersimpan dapat diakses, diubah, atau dihapus sesuai kebutuhan aplikasi melalui *layer repository*.

3.3.2 *EMS-Service*

EMS-Service adalah layanan *microservice* berbasis Spring Boot yang berfungsi sebagai penghubung antara sistem EMS (*Event Monitoring System*) dan layanan internal (APICenter), dengan tujuan utama meneruskan dan mencatat data kejadian (*event*) terkait kondisi gangguan atau anomali pada suatu sistem atau mesin. Data yang dicatat seperti *condition, description,* dan *time* terjadinya gangguan ke *database gateway* agar dapat dipantau secara real-time. Dibangun dengan arsitektur *layered* (*controller, service, repository,* dan *integration layer*), layanan ini memastikan modularitas, pemisahan tanggung jawab, serta integrasi yang terukur, dan menghilangkan ketergantungan pada proses manual. Meskipun EMS-Service dibangun menggunakan pendekatan *layered architecture* secara

internal, layanan ini di-*deploy* sebagai unit mandiri dalam ekosistem *microservice* yang terdiri dari layanan-layanan seperti *Auth-Service, API Center*, dan *Gateway*.

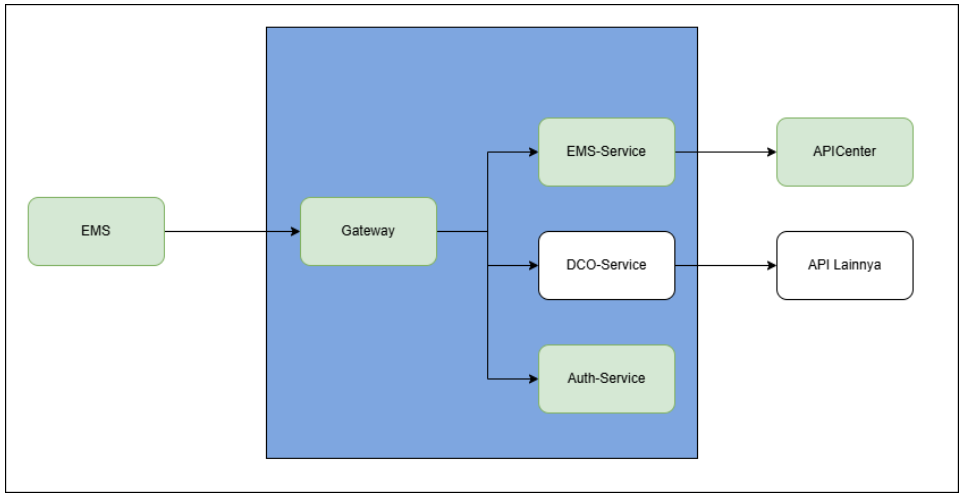
A Konsep Dasar

EMS-Service dikembangkan dalam ekosistem *microservices*, yaitu di mana sistem dibagi menjadi layanan-layanan kecil(*Service*) yang dapat berjalan secara *independent* namun tetap saling terhubung melalui protokol komunikasi ringan seperti HTTP.

Microservices adalah gaya arsitektur yang membangun sebuah aplikasi sebagai kumpulan layanan yang terpisah, dimana setiap layanan menyediakan satu bagian dari logika bisnis. Dan *microservice* merupakan versi modular dari arsitektur monolitik dan dirancang agar setiap layanan bersifat otonom dan independen, serta dapat di*deploy* secara terpisah [11].

Kelebihan *microservice* adalah mudah dalam pemeliharaan karena setiap fungsi utama merupakan modul yang terpisah, serta lebih andal karena gangguan pada satu *microservice* tidak mempengaruhi seluruh sistem, dan mudah untuk diskalakan [11].

Berikut ini merupakan gambaran arsitektur sistem *EMS-Service* yang dikembangkan selama masa magang:



Gambar 3.3. Arsitektur *EMS-Service* dalam *Microservices*
Sumber : [9]

Pada Gambar 3.3, masing-masing *service* memiliki peran yang berbeda-beda, yaitu:

1. EMS (*Event Monitoring System*)
   EMS merupakan sistem internal yang melakukan pengiriman data *request* ke sistem *backend*. Data yang dikirim merupakan informasi yang akan diproses dan diteruskan ke layanan internal (*APICenter*).

2. *Gateway* (API *Gateway Layer*)
   Seluruh *request* dari EMS akan masuk terlebih dahulu ke *layer Gateway*. *Gateway* berfungsi sebagai *entry point* tunggal bagi seluruh *request* yang masuk, dan akan menentukan *service* tujuan berdasarkan *routing path*. *Gateway* juga memiliki peran penting dalam *logging, rate limiting*, serta pengamanan awal saat dilakukan *request*.

3. *Auth-Service* (*Authentication Service*)
   Sebelum *request* diteruskan ke *service* tujuan, *Gateway* terlebih dahulu mengarahkan *request* ke *Auth-Service* untuk *login* dan menghasilkan token *autentikasi* (JWT). Token ini wajib disertakan oleh EMS saat melakukan *request* ke *EMS-Service* agar dapat diverifikasi oleh *Gateway* atau *EMS-Service* sebelum diproses lebih lanjut.

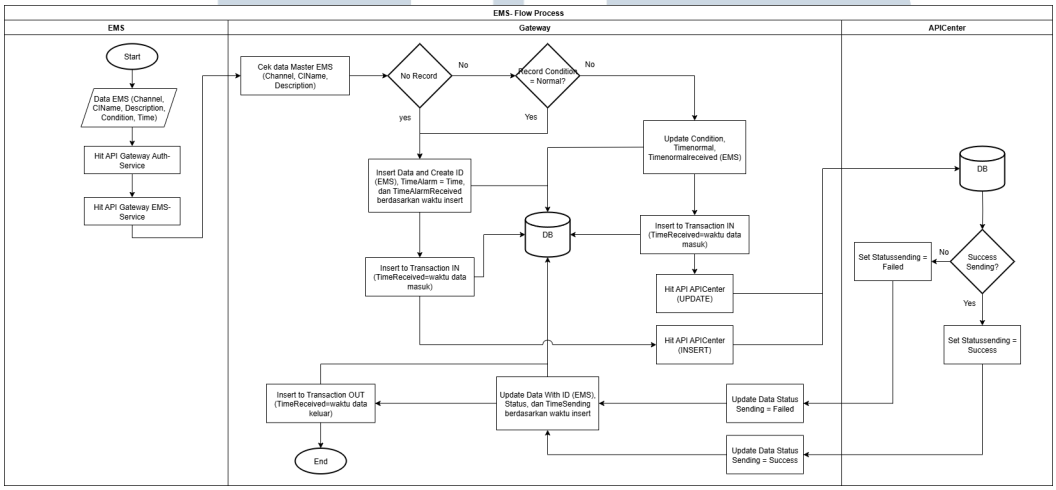
4. *EMS-service* (*Service* utama)
   EMS-Service merupakan komponen utama dalam sistem ini yang bertugas untuk menyiapkan *data request*, mencatat histori permintaan, dan meneruskan *request* yang telah lolos autentikasi dari EMS ke layanan *service* utama, yaitu *APICenter*. Seluruh proses dilakukan secara terstruktur agar setiap *request* dapat ditelusuri dan sesuai standar integrasi sistem.

5. *APICenter*
   APICenter merupakan layanan internal pusat yang menjadi tujuan akhir dari *request* yang dikirim oleh EMS melalui *EMS-Service*. *EMS-Service* bertanggung jawab memastikan data diteruskan sesuai spesifikasi dan menunggu *response* untuk diteruskan kembali ke EMS.

Pada sistem arsitektur *microservices* yang ditunjukkan pada Gambar 3.3, *EMS-Service* merupakan satu-satunya layanan yang dikembangkan dalam kegiatan magang ini. Komponen lain seperti *EMS, Gateway, Auth-Service*, dan *APICenter* merupakan bagian dari sistem internal yang telah tersedia sebelumnya di lingkungan PT Bank Central Asia Tbk. Ruang lingkup pengembangan terbatas pada implementasi dan penyempurnaan *EMS-Service* sebagai layanan penghubung antarsistem dalam ekosistem tersebut.

B *Flow Process* EMS-Service

Setelah memahami *EMS-Service* dalam ekosistem *microservices*, berikutnya adalah melihat secara lebih rinci bagaimana alur proses data berjalan dari awal hingga akhir. *Flowchart* berikut menggambarkan proses bisnis dan teknis secara berurutan, dimulai dari sistem EMS sebagai sumber data, kemudian diproses oleh *Gateway* dan *EMS-Service*, hingga akhirnya diteruskan ke layanan internal (*APICenter*).



Gambar 3.4. *Flow Process EMS*

Sumber : [9]

Berdasarkan Gambar 3.4, Proses dimulai dari sistem EMS yang mengirimkan data berisi parameter seperti *Channel, CIName, Description, Condition*, dan *Time*. Data ini pertama-tama digunakan untuk melakukan autentikasi melalui pemanggilan API *Auth-Service* guna memperoleh token akses. Setelah token berhasil diperoleh, EMS akan melakukan *request* ke *Gateway EMS-Service* dengan menyertakan token tersebut.

Saat request masuk ke *Gateway*, sistem akan memeriksa apakah kombinasi data (*Channel, CIName*, dan *Description*) sudah tersedia dalam *master* data EMS. Jika tidak ditemukan (*No Record*), maka sistem akan membuat ID baru dan mencatat waktu *TimeAlarm* dan *TimeAlarmReceived*, lalu menyimpan data ke database, serta mencatatnya sebagai transaksi masuk (*Transaction IN*).

Jika data ditemukan, *Gateway* akan memeriksa apakah kondisi data tersebut "Normal". Bila ya, maka sistem akan membuat ID baru dan mencatat ulang waktu *TimeAlarm* dan *TimeAlarmReceived*. Namun jika kondisi tidak normal, sistem akan memperbarui informasi kondisi (*Condition*), waktu normal (*TimeNormal*), dan

waktu terjadinya perubahan (*TimeNormalReceived*). Setelahnya, data disimpan ke database dan juga dicatat ke *Transaction IN*.

Dari titik ini, *EMS-Service* akan melakukan pemanggilan ke *APICenter* melalui dua jenis *request*, yaitu *UPDATE* untuk data yang sebelumnya sudah pernah dikirim, dan *INSERT* untuk data baru yang belum ada di sistem *APICenter*.

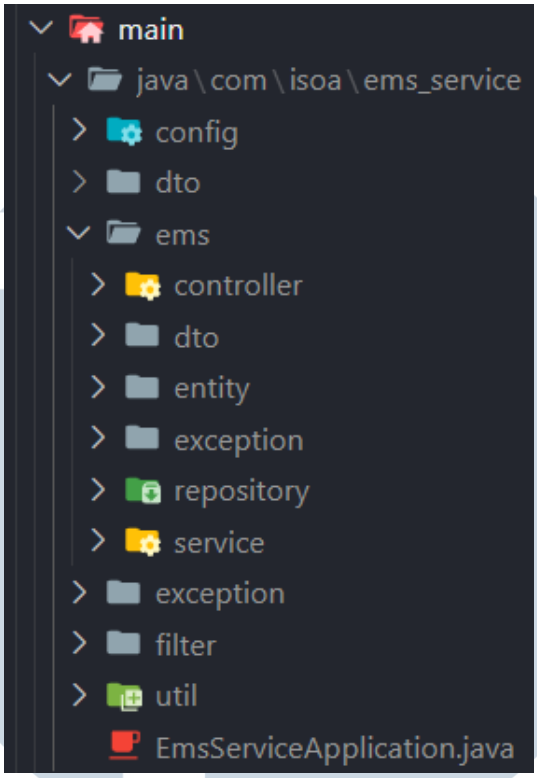
APICenter kemudian menyimpan data ke dalam *database*nya dan melakukan validasi apakah pengiriman data berhasil. Jika berhasil, maka status *Sending* akan diperbarui menjadi *Success*; jika gagal, status akan diset ke *Failed*.

Terakhir, *EMS-Service* akan melakukan *update* data transaksi dengan ID EMS, status pengiriman (*Sending*), dan waktu pengiriman (*TimeSending*). Proses ditutup dengan mencatat data sebagai transaksi keluar (*Transaction OUT*), sebagai penanda bahwa data telah selesai diproses.

C Implementasi

EMS-Service dikembangkan menggunakan *framework Spring Boot* dan dirancang berdasarkan prinsip arsitektur *microservices*. Implementasi *service* ini melibatkan beberapa komponen penting seperti *controller*, DTO, *entity*, *repository*, *service*, *exception handling*, dan konfigurasi pendukung lainnya.

Struktur folder *EMS-Service* mengikuti standar konvensi *Spring Boot*, yang memisahkan setiap lapisan kode berdasarkan tanggung jawabnya. Gambar 3.5 menunjukkan struktur folder dari proyek *EMS-Service*.



Gambar 3.5. Struktur Folder

Sumber : [9]

Folder *controller* berisi *class* yang menangani *request* HTTP dari *client*. Salah satu contoh implementasi *controller* adalah *endpoint POST /*ems, yang menerima data dari EMS, memvalidasi *input*, dan meneruskannya ke *service layer* untuk diproses lebih lanjut. Data yang diterima dan dikembalikan dikemas menggunakan *class* DTO (*Data Transfer Object*) seperti *GlobalResponseDto*, *OutputSchemaResponseDto*, dan *OutputSchemaDataResponseDto*, yang diletakkan pada folder dto.

Entity dalam *EMS-Service* merepresentasikan struktur tabel di *database*, misalnya entitas transaksi yang mencatat waktu masuk (*timeReceived*), waktu kirim (*timeSending*), status pengiriman, dan detail data EMS lainnya. Entity ini dikelola menggunakan *Spring Data* JPA melalui *interface repository* yang diletakkan dalam folder *repository*, seperti *EmsTransactionRepository*.

Logika utama bisnis berada di dalam *folder service*. Di sinilah proses seperti pengecekan master data, pencatatan data masuk dan keluar, pemilihan metode *INSERT* atau *UPDATE* ke *APICenter*, serta pencatatan status pengiriman dilakukan. Seluruh proses ini dikemas dalam *method* yang diakses dari *controller*.

Untuk menangani kesalahan, *EMS-Service* menyediakan folder *exception*

yang berisi *class custom exception* dan *handler global* berbasis anotasi *@ControllerAdvice*. Dengan demikian, semua error yang terjadi di dalam *service* dapat dikembalikan dalam format JSON yang konsisten dan mudah ditangani oleh sistem pemanggil.

Selain komponen utama di atas, terdapat pula *folder config, filter,* dan *util*. *Folder config* digunakan untuk menyimpan konfigurasi *global* seperti *bean RestTemplate*, sementara *util* digunakan untuk menyimpan *helper class* (misalnya *object AppMessage*). Jika diterapkan, folder filter dapat digunakan untuk *logging request* masuk atau validasi token.

Terakhir, file *EmsServiceApplication* merupakan kelas utama *Spring Boot* yang menjalankan keseluruhan aplikasi. Kelas ini ditandai dengan anotasi *@SpringBootApplication* dan menjadi titik awal eksekusi saat aplikasi dijalankan.

Seluruh struktur dan implementasi ini saling bekerja sama untuk menjalankan fungsi *EMS-Service* sesuai alur yang dijelaskan pada bagian *flowchart* sebelumnya.

1. *Controller*

   Salah satu komponen utama dari *EMS-Service* adalah *controller*, yang bertugas menangani permintaan (*request*) dari *client*, dalam hal ini *Gateway* yang mewakili sistem EMS. *Controller* menggunakan anotasi *@RestController* dan *@RequestMapping* dari *Spring Boot* untuk mendefinisikan *endpoint* HTTP. Pada *EMS-Service*, endpoint yang digunakan adalah *POST* /ems, yang menerima *request body* dalam bentuk JSON dan memetakan datanya ke objek DTO (*EmsRequestDto*).

   Contoh implementasi *controller* dapat dilihat pada Kode 3.1 berikut:

```
@RestController
@Tag(name = "EMS", description = "Endpoints for EMS")
@RequestMapping("ems")
@Slf4j
public class EMSController {

    private final MessageUtils messageUtils;
    private final APICenterService apicenterservice;

    public EMSController(MessageUtils messageUtils,
    APICenterService apicenterservice) {
        this.messageUtils = messageUtils;
        this.apicenterservice = apicenterservice;
    }

    @Operation(summary = "Post New EMS Transaction")
    @PostMapping
    public ResponseEntity<GlobalResponseDto<
    OutputSchemaDataResponseDto<APICenterResponseDTO>>>
    createNewEMS(@Valid @RequestBody MasterEMSRequest
    masterEMSRequest){
```

```
19          GlobalResponseDto<OutputSchemaDataResponseDto<
    APICenterResponseDTO>> responseDto =
20                  messageUtils.successWithParamDto(
21                      apicenterservice.senddatatoapicenter(
    masterEMSRequest),
22                      AppErrorEnum.CREATED,
23                      AppMessageEnum.EMS);
24              return new ResponseEntity<>(responseDto, HttpStatus.
    OK);
25          }
26  }
```

Kode 3.1: Contoh implementasi *Controller* pada *EMS-Service*

Pada Kode 3.1 menunjukkan bahwa *controller* menerima *request* dalam bentuk objek *masterEMSRequest*, kemudian meneruskannya ke *layer service* (*apicenterservice*) untuk diproses. Hasil pemrosesan akan dikembalikan dalam bentuk objek *GlobalResponseDto* sebagai *response* HTTP dengan status kode 200 (OK).

Penggunaan anotasi *@Valid* menandakan bahwa *input* dari *client* akan divalidasi terlebih dahulu sesuai dengan aturan yang ditentukan di dalam *class masterEmsRequest*. Dengan struktur seperti ini, *controller* hanya fokus pada menerima dan meneruskan *request*, sementara logika bisnis didelegasikan sepenuhnya ke *layer service*, sesuai dengan prinsip *separation of concerns* dalam pengembangan aplikasi berbasis *Spring Boot*.

2. DTO (*Data Transfer Object*)

   Untuk memisahkan antara struktur data yang diterima dari *client* dan yang dikembalikan sebagai *response*, *EMS-Service* menggunakan beberapa class DTO (*Data Transfer Object*). DTO berfungi sebagai objek perantara yang menyederhanakan dan mengamankan proses pertukarn data antara *client* dan *service*, serta menghindari pemetaan langsung ke *entity database*.

   Pada *EMS-Service*, terdapat dua jenis DTO yang digunakan yaitu *requestDTO* dan *responseDTO*. Salah satu DTO utama yang digunakan untuk permintaan masuk adalah *masterEMSRequest*, yang merepresentasikan data yang dikirim oleh sistem EMS. DTO ini terdiri dari *field* seperti *channel, ciName, description, condition*, dan *time*, serta dapat diberikan anotasi validasi seperti *@NotNull*, *@Size*, atau *@Pattern* untuk memastikan input *valid* sebelum diproses lebih lanjut. Contoh implementasi *RequestDTO* dapat dilihat pada Kode 3.2 berikut:

```
1  @Builder
2  @Data
3  @AllArgsConstructor
4  @NoArgsConstructor
5  @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
6  public class MasterEMSRequest {
```

```
7
8       @NotBlank(message = "Channel is required and must not be
        blank")
9       private String channel;
10
11      @NotBlank(message = "CIName is required and must not be
        blank")
12      private String ciName;
13
14      @NotBlank(message = "Description is required and must not
         be blank")
15      private String description;
16
17      @NotBlank(message = "Condition is required and must not
        be blank")
18      @ValidCondition
19      private String condition;
20
21      private String time;
22  }
```

Kode 3.2: Contoh implementasi *Request* DTO pada *EMS-Service*

Untuk *response*, *EMS-Service* menggunakan objek seperti *GlobalResponseDTO*, *OutputSchemaResponseDTO*, dan *OutputSchemaDataResponseDTO*. Salah satu contoh *response* standar yang dikembalikan ke *client* adalah *GlobalResponseDTO*, yang menyertakan informasi *errorschema, outputschema, errorcode*, dan *errormessage*. Contoh implementasi *ResponseDTO* dapat dilihat pada Kode 3.3 berikut:

```
1  @Builder
2  @Data
3  @AllArgsConstructor
4  @NoArgsConstructor
5  @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
6  public class GlobalResponseDto<T> {
7
8      private ErrorSchema errorSchema;
9      private T outputSchema;
10
11     @Builder
12     @Data
13     @AllArgsConstructor
14     @NoArgsConstructor
15     @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.
       class)
16     public static class ErrorSchema {
17         private String errorCode;
18         private ErrorMessage errorMessage;
19     }
20
21     @Builder
22     @Data
23     @AllArgsConstructor
24     @NoArgsConstructor
25     public static class ErrorMessage {
26         private Object indonesian;
27         private Object english;
28     }
29 }
```

Kode 3.3: Contoh implementasi *Response DTO* pada *EMS-Service*

Dengan penggunaan DTO, *EMS-Service* dapat menjaga struktur *response* agar tetap konsisten, mudah di-*debug*, dan aman untuk menghindari pemetaan langsung ke *entity database*.

3. *Entity* Dalam *EMS-Service*, *entity* digunakan untuk merepresentasikan struktur data yang akan disimpan ke dalam *Database*. *Entity* adalah *class java* yang dipetakan ke tabel dalam *database* menggunakan anotasi JPA (*Java Persistence API*), seperti *@Entity, @Table, @Column*, dan sebagainya.Setiap objek *entity* akan diolah oleh *repository* untuk proses penyimpanan, pembaruan, atau pengambilan data dari *database* PostgreSQL. Salah satu contoh *entity* utama dalam *EMS-Service* adalah *TransactionEMS Entity*, yang menyimpan informasi terkait request *IN* dan *OUT* dari EMS. *Field* dalam entitas ini mencakup berbagai data penting seperti *channel, ciName, description, condition, timeReceived, timeSending, status*, dan lainnya. Informasi ini digunakan untuk mencatat histori transaksi, memantau status pengiriman ke *APICenter*, serta mendukung fitur pelacakan (*tracking*) data. Contoh implementasi *Entity* dapat dilihat pada Kode 3.4 berikut:

```
@Entity
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "TR_EMS")
public class TransactionEMS {

    @Id
    @Column(name = "id")
    private UUID id;

    @ManyToOne
    @JoinColumn(name = "master_id", referencedColumnName = "
    master_id")
    private MasterEMS masterEMS;

    @Column(name = "channel")
    private String channel;

    @Column(name = "ciname")
    private String ciname;

    @Column(name = "description")
    private String description;

    @Column(name = "condition")
    private String condition;

    @Column(name = "statusSending")
    private String statusSending;

    @Column(name = "time")
    private LocalDateTime time;

    @Column(name = "time_received")
    private LocalDateTime timeReceived;

```

```
38      @Column(name = "action")
39      private String action;
40
41      @Column(name = "eventNumber")
42      private String eventNumber;
43  }
```

Kode 3.4: Contoh implementasi *Entity* pada EMS-Service

Entity ini secara otomatis akan dipetakan oleh *Spring Data* JPA ke dalam tabel *Transaction* di database. Dengan pendekatan ini, proses penyimpanan dan pengambilan data dapat dilakukan dengan lebih efisien, aman, dan terintegrasi dengan baik ke dalam lapisan *service*.

4. *Repository*

   Untuk mengelola proses penyimpanan, pengambilan, dan pembaruan data pada *database*, *EMS-Service* menggunakan komponen *repository* yang dibangun dengan *Spring Data* JPA. *Repository* adalah antarmuka (*interface*) yang berfungsi sebagai penghubung antara aplikasi dan basis data, memungkinkan interaksi dengan *entity* tanpa perlu menulis perintah SQL secara eksplisit.

   Pada *EMS-Service*, salah satu *repository* yang digunakan adalah *MasterRepository*, yang bertanggung jawab untuk mengakses data dari tabel *master*. *Interface* ini meng-*extend JpaRepository*, sehingga secara otomatis menyediakan berbagai method standar seperti *save(), findById(), findAll()*, dan *deleteById*().

```
1  @Repository
2  public interface MasterRepositoy extends JpaRepository<
       MasterEMS, String> {
3
4       @Query("SELECT m FROM MasterEMS m WHERE " +
5               "LOWER(TRIM(m.channel)) = LOWER(TRIM(:channel))
       AND " +
6               "LOWER(TRIM(m.ciName)) = LOWER(TRIM(:ciName)) AND
       " +
7               "LOWER(TRIM(m.description)) = LOWER(TRIM(:
       description)) AND " +
8               "LOWER(TRIM(m.condition)) != 'normal'")
9       Optional<MasterEMS>
       findActiveByChannelAndCiNameAndDescription(
10          @Param("channel") String channel,
11          @Param("ciName") String ciName,
12          @Param("description") String description
13      );
14
15      Optional<MasterEMS> findBymasterId(UUID MasterId);
16      List<MasterEMS> findAll();
17      List<MasterEMS> findByStatusSendingIgnoreCase(String
       statusSending);
18  }
```

Kode 3.5: Contoh implementasi *Repository* pada EMS-Service

Pada Kode 3.5 menunjukkan bahwa selain *method* bawaan dari *JpaRepository*, *repository* juga dapat memiliki *custom query method* yang disesuaikan dengan kebutuhan logika bisnis, seperti pencarian berdasarkan kombinasi *channel, ciName,* dan *description*. Hasil dari *query* ini dibungkus dalam *Optional*, untuk mengantisipasi kemungkinan data tidak ditemukan.

Dengan penggunaan *Spring Data JPA*, *repository* dapat mengelola data secara efisien dan terintegrasi langsung dengan *entity* dan *service layer*, tanpa harus menulis kode SQL secara manual, sehingga mendukung pengembangan sistem yang lebih cepat dan terstruktur.

5. *Service*

   Service dalam *EMS-Service* berperan sebagai pusat dari logika bisnis utama. Salah satu *service* penting adalah *APICenterServiceImpl*, yang bertanggung jawab untuk memproses data EMS dan meneruskannya ke *Service* lainnya, yaitu *APICenter*, melalui *HTTP request*.

   Class ini diimplementasikan menggunakan anotasi *@Service* dan mengimplementasikan *interface APICenterService*. Metode utama dalam *class* ini adalah *sendRequestToAPICenter*(), yang menerima objek *APICenterRequest*, menyiapkan *header* (termasuk *authorization key* dan *content type*), mencatat waktu *request* (*timeReceived*), dan melakukan pemanggilan ke *endpoint APICenter* menggunakan *RestTemplate*.

```
@Service
@Slf4j
public class APICenterServiceImpl implements APICenterService
    {

    @Value("${APICenter.url}")
    private String url;

    @Value("${APICenter.api-key}")
    private String apikey;

    private final MasterEMSRepository masterEMSRepository;
    private final APICenterService apicenterservice;
    private final TransactionEMSService transactionEMSService
   ;

    @Autowired
    RestTemplate restTemplate = new RestTemplate(
        new SimpleClientHttpRequestFactory() {{
            setConnectTimeout(60000);
            setReadTimeout(60000);
        }}
    );

    public APICenterServiceImpl(MasterEMSRepository
   masterEMSRepository,
                                APICenterService
   apicenterservice,
```

```
                                    TransactionEMSService
    transactionEMSService) {
        this.masterEMSRepository = masterEMSRepository;
        this.apicenterservice = apicenterservice;
        this.transactionEMSService = transactionEMSService;
    }

    public APICenterResponseDTO sendRequestToAPICenter(
    APICenterRequest request) {
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        headers.set("Authorization", apikey);
        LocalDateTime timeReceived = getCurrentDateTime();

        HttpEntity<APICenterRequest> requestEntity = new
    HttpEntity<>(request, headers);

        try {
            ResponseEntity<String> response = restTemplate.
    exchange(
                url,
                HttpMethod.POST,
                requestEntity,
                String.class
            );

            String responseBody = response.getBody();
            ObjectMapper objectMapper = new ObjectMapper();
            APICenterResponseDTO responseDTO = objectMapper.
    readValue(responseBody, APICenterResponseDTO.class);

            log.info("Response Body from APICenter: {}",
    responseBody);
            updateMasterEMSStatus(request.getDataEvent().
    getBcaidEvent(), responseDTO, timeReceived);

            return responseDTO;

        } catch (ResourceAccessException e) {
            log.error("Timeout or connection error to
    APICenter, Status be Pending", e);
            updateMastersEMSStatusToPending(request.
    getDataEvent().getidEvent());
            throw new RuntimeException("Timeout or connection
     error to APICenter, Status be Pending");

        } catch (Exception e) {
            log.error("Failed to send data to APICenter", e);
            updateMasterEMSStatusToPending(request.
    getDataEvent().getidEvent());
            throw new RuntimeException("Failed to send data
    to APICenter");
        }
    }

    public APICenterRequest sendToMasterEMS(MasterEMSRequest
    masterEMSRequest) {
        return masterEMSService.createEMSMaster(
    masterEMSRequest);
    }
}
```

Kode 3.6: Contoh implementasi *Service* pada EMS-Service

Pada Kode 3.6 terlihat bahwa *service* ini juga menangani pengecualian

(*exception*) untuk dua skenario: *timeout/connection error* dan *error* umum lainnya. Pada kedua kondisi tersebut, *status request* akan diset ke *pending* agar dapat ditindaklanjuti ulang. Proses ini menunjukkan bahwa sistem dirancang untuk resilien terhadap gangguan jaringan atau error eksternal.

Dengan struktur ini, lapisan *service* berhasil merangkum logika teknis seperti pengiriman *HTTP request*, pemrosesan *response, logging*, dan *fallback mechanism* dalam satu *class*, yang membuat *controller* mudah diuji.

6. *Exception*

   Selain *controller, DTO, entity, repository,* dan *service*, *EMS-Service* juga menerapkan mekanisme *exception handling* secara *global* untuk menangani kesalahan yang terjadi selama proses eksekusi. Hal ini dilakukan agar sistem dapat memberikan *response* yang terstruktur, mudah ditelusuri, dan tetap aman saat terjadi *error*.

   EMS-Service menggunakan *class GlobalExceptionHandler* yang diberi anotasi @*ControllerAdvice* untuk menangani semua *exception* yang tidak tertangani secara eksplisit. Di dalamnya terdapat method *handleGlobalException*() yang menangani semua jenis *exception* umum (*Exception.class*) dan mengembalikan *response* dalam format *GlobalResponseDto* dengan status HTTP 500 (*INTERNAL SERVER ERROR*).

```
@ControllerAdvice
public class GlobalExceptionHandler {

    private final MessageUtils messageUtils;

    public GlobalExceptionHandler(MessageUtils messageUtils)
    {
        this.messageUtils = messageUtils;
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<GlobalResponseDto<Object>>
    handleGlobalException(Exception ex) {
        LoggingAspect.printException("Global", "Handler", ex)
    ;
        GlobalResponseDto<Object> responseDto = messageUtils.
    globalExceptionDto(ex);
        return new ResponseEntity<>(responseDto, HttpStatus.
    INTERNAL_SERVER_ERROR);
    }
}
```

Kode 3.7: Contoh implementasi *Exception* pada EMS-Service

Pada Kode 3.7, *exception* yang tertangkap dicatat melalui *LoggingAspect.printException*() dan dikonversi menjadi *response* yang

dapat dibaca oleh *client* menggunakan bantuan *class MessageUtils*. Hasilnya adalah struktur JSON yang konsisten, berisi informasi seperti *status*, pesan kesalahan, dan *timestamp*, tanpa mengekspos *stack trace* ke luar sistem.

7. Application

   Sebagai aplikasi berbasis Spring Boot, *EMS-Service* memiliki satu kelas utama yang bertanggung jawab menjalankan seluruh aplikasi. *class* ini bernama *EmsServiceApplication* dan berada pada akar struktur proyek. Kelas ini ditandai dengan anotasi *@SpringBootApplication*, yang merupakan kombinasi dari tiga anotasi inti Spring: *@Configuration*, *@EnableAutoConfiguration*, dan *@ComponentScan*. Ketiganya secara otomatis mengatur konfigurasi aplikasi dan pemindaian komponen saat proses startup.

```
@SpringBootApplication
public class EmsServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(EmsServiceApplication.class,
    args);
    }

}
```

Kode 3.8: Contoh implementasi *Application Spring Boot* pada EMS-Service

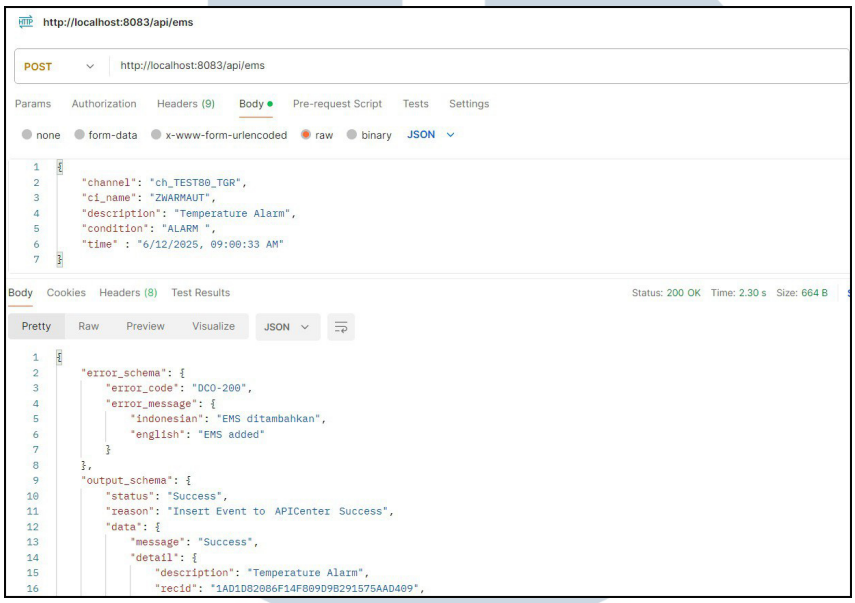
Pada Kode 3.8, Method *main*() merupakan titik masuk (*entry point*) dari aplikasi. Saat dijalankan, *SpringApplication.run*() akan menginisialisasi konteks *Spring*, memuat semua *konfigurasi*, dan menyiapkan seluruh komponen yang dibutuhkan aplikasi, termasuk *controller, service*, dan *repository*. Struktur ini memungkinkan *EMS-Service* untuk dijalankan secara independen sebagai *microservice* dan siap menerima *request* dari sistem eksternal melalui *Gateway*.

D Hasil Implementasi

Setelah proses implementasi selesai, pengujian dilakukan menggunakan Postman untuk memastikan bahwa *endpoint EMS-Service* berfungsi dengan benar dan mampu memproses *request* dari EMS serta meneruskannya ke *APICenter*. Pengujian dilakukan terhadap skenario utama, yaitu: *request* dengan data baru, *request* dengan data yang sudah ada, dan *request* yang gagal karena koneksi ke *APICenter*.

1. Pengujian Data *Valid* (*Success*)

   Pengujian dilakukan terhadap *endpoint POST* /ems dengan mengirimkan data yang *valid* dari sistem EMS. Data dikirim dalam *format* JSON, yang berisi atribut seperti *channel, ciname, description, condition*, dan *time*.



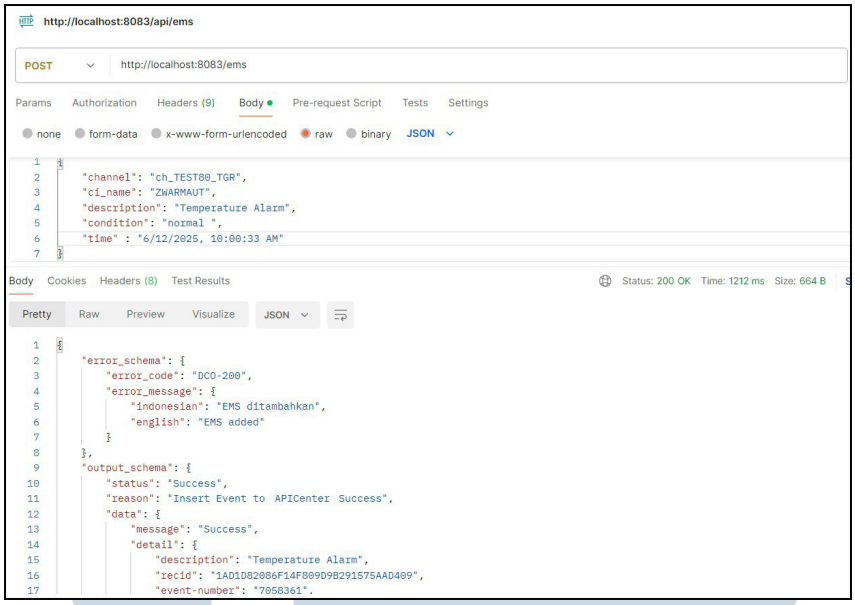
Gambar 3.6. Pengujian Data *Valid* (*Success*)

Sumber : [9]

Pada Gambar 3.6, Setelah *request* dikirim, *EMS-Service* berhasil memproses data dan meneruskannya ke *APICenter*. *Response* yang diterima dari *server* memiliki status HTTP 200 *OK*. Isi *response* dikembalikan dalam format JSON dengan dua bagian utama yaitu *errorschema* dan *outputschema*

2. Pengujian *Data Update* Dengan *Condition Normal*

   Pengujian ini bertujuan untuk memastikan bahwa *EMS-Service* dapat menangani kondisi ketika data yang dikirim sudah pernah tercatat sebelumnya, namun kali ini dalam *status normal*. Data tersebut seharusnya dikirim ulang ke *APICenter* dengan *action create* tanpa membuat *ID Master* baru, tetapi cukup diperbarui *actionnya* di *internal* sistem dan kirim format update ke APICenter.



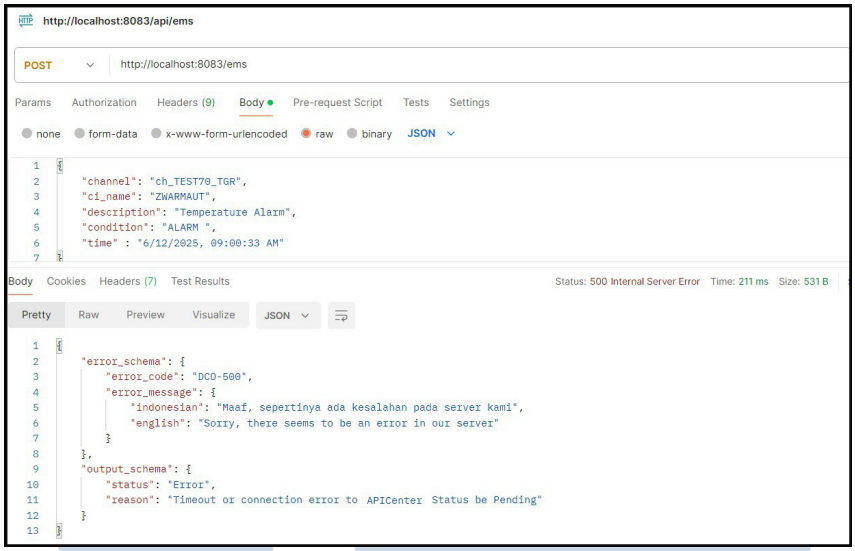
Gambar 3.7. Pengujian Data Update

Sumber : [9]

Berdasarkan Gambar 3.7, *Response* yang dikembalikan menunjukkan *status success* atau berhasil. Hal ini ditunjukkan dengan nilai *event-number* pada *response* yang sama dengan *event* sebelumnya, menandakan bahwa sistem tidak membuat entri baru, melainkan hanya memperbarui *status* dari data yang sudah ada.

3. Pengujian Koneksi Gagal (*Status Pending*)

   Pengujian ini dilakukan untuk mensimulasikan kondisi ketika *EMS-Service* tidak dapat meneruskan *request* ke *APICenter* karena terjadi masalah koneksi, seperti *timeout* atau kesalahan pada *server* tujuan. Dalam implementasinya, *EMS-Service* harus tetap merespons dengan baik, menandai data sebagai *pending*, dan tidak menyebabkan *crash* sistem.

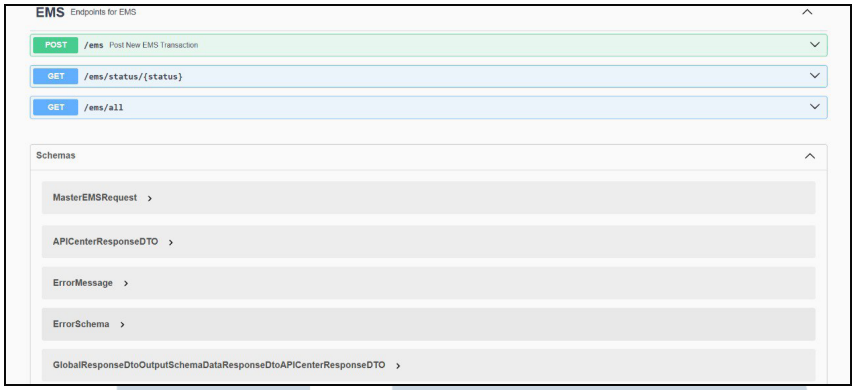


Gambar 3.8. Pengujian status Pending

Sumber : [9]

Pada Gambar 3.8, Respons ini menunjukkan bahwa *EMS-Service* telah menangani *error* dengan baik melalui mekanisme *fallback*, yaitu dengan tidak memaksakan koneksi ulang yang dapat menambah beban sistem, serta menyimpan status transaksi sebagai *pending* untuk memungkinkan proses ulang (*retry*) atau penanganan lanjutan secara manual maupun otomatis. Pengujian ini membuktikan bahwa sistem memiliki ketahanan (*resilience*) yang baik terhadap gangguan eksternal, dan bahwa mekanisme penanganan *exception* telah berjalan sesuai dengan implementasi yang terdapat pada *GlobalExceptionHandler*.

Setelah seluruh skenario pengujian berhasil dijalankan dan hasilnya sesuai dengan ekspektasi yang diharapkan, berikutnya adalah melakukan implementasi dokumentas API menggunakan Swagger. Dokumentasi ini bertujuan untuk memastikan bahwa setiap endpoint yang tersedia terdokumentasi dengan baik dan sesuai dengan fungsionalitas aslinya.



Gambar 3.9. Dokumentasi Swagger pada EMS-Service

Sumber : [9]

Berdasarkan Gambar 3.9, Swagger menampilkan setiap endpoint yang terdapat dalam EMS-Service, dan juga menampilkan skema data (DTO) yang digunakan dalam komunikasi antar sistem. Seluruh struktur ini membantu tim *developer* dan tim QA dalam memahami dan menguji sistem tanpa harus membaca kode sumber secara langsung.

Dengan adanya dokumentasi Swagger ini, EMS-Service siap untuk diintegrasikan secara penuh dengan layanan internal lannya, serta dapat digunakan dan di uji secara lebih efisien dan transparan.

3.4 Kendala dan Solusi yang Ditemukan

3.4.1 Kendala

1. Selama proses testing secara lokal, terjadi kendala akibat pembatasan *proxy* perusahaan yang menyebabkan munculnya *error PKIX Certification* saat mencoba mengakses *API APICenter*.
2. Sistem *Backend* yang dikembangkan berada dalam lingkungan intranet perusahaan, sehingga hanya dapat diakses melalui jaringan internal kantor. Sementara itu, perangkat laptop yang disediakan tidak memiliki akses ke jaringan tersebut saat berada di luar kantor, yang menyebabkan kendala dalam proses penyusunan dan pengujian aplikasi saat bekerja secara *remote*.

3.4.2 Solusi

1. Untuk mengatasi kendala teknis, solusi yang diterapkan adalah dengan menggunakan *UnsafeRestTemplate* saat melakukan *testing* di lingkungan

lokal, sehingga *bypass* terhadap sertifikat dapat dilakukan dan pengujian terhadap *API APICenter* tetap bisa berjalan dengan lancar.

2. Dalam menghadapi keterbatasan akses jaringan, peserta magang memanfaatkan layanan cloud storage untuk menyimpan referensi dan dokumen penting yang dibutuhkan dalam penyusunan laporan. Hal ini memungkinkan proses penyusunan tetap dapat dilanjutkan dengan perangkat pribadi di luar lingkungan kantor.