BAB 3
PELAKSANAAN KERJA MAGANG

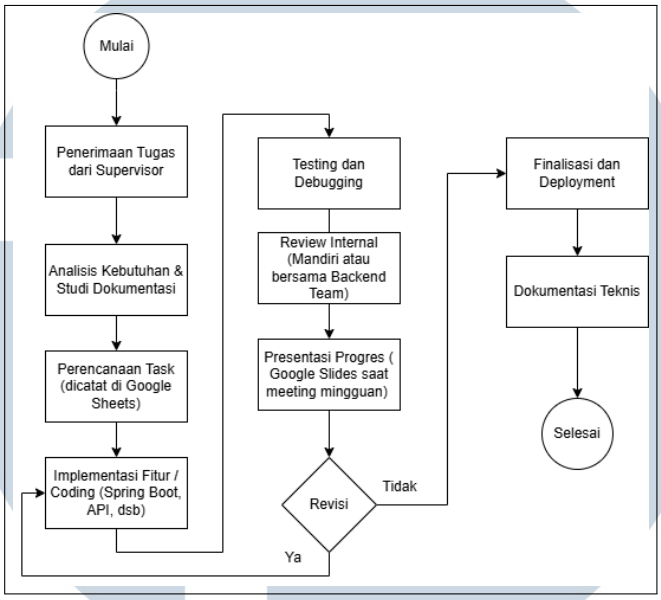
3.1 Kedudukan dan Koordinasi

Magang dilaksanakan di PT Jaya Santoso Teknologi sebagai *Backend Intern* dan bergabung dalam tim Divisi Pengembangan IT yang bertanggung jawab terhadap integrasi layanan pesan di berbagai platform. Pelaksanaan kegiatan magang dilakukan dengan kedudukan sebagai *Backend Intern* yang berada di bawah supervisi *Backend Developer*, dengan rincian alur kerja, koordinasi, dan kedudukan tersebut dijelaskan pada subbagian di bawah ini.

3.1.1 Alur Kerja *Backend Intern*

Selama pelaksanaan magang di PT Jaya Santoso Teknologi, alur kerja yang diikuti oleh *Backend Intern* bersifat terstruktur dan sistematis. Proses kerja dimulai dengan penerimaan tugas dari *supervisor* yang bertanggung jawab memberikan arahan teknis. Setelah mendapatkan tugas, langkah berikutnya adalah melakukan analisis kebutuhan serta studi terhadap dokumentasi teknis yang relevan. Hal ini bertujuan untuk memahami ruang lingkup pengembangan dan memastikan implementasi dilakukan sesuai spesifikasi. Selanjutnya, dilakukan perencanaan tugas secara rinci yang dicatat pada Google Sheets. Perencanaan ini mencakup pembagian tugas menjadi bagian-bagian yang lebih kecil agar proses pengerjaan menjadi lebih terukur. Setelah itu, tahap implementasi dilakukan, yaitu pengembangan fitur atau layanan menggunakan Spring Boot, termasuk pembuatan API, integrasi sistem, dan pengelolaan basis data. Setelah implementasi selesai, dilanjutkan dengan proses *testing* dan *debugging* untuk memastikan sistem berjalan dengan baik dan bebas dari kesalahan. Hasil kerja kemudian direview secara internal, baik secara mandiri maupun bersama anggota tim *backend*. Progres pengembangan juga dipresentasikan dalam pertemuan mingguan menggunakan Google Slides, sebagai sarana evaluasi dan pemberian umpan balik. Apabila ditemukan kekurangan atau ketidaksesuaian, maka dilakukan revisi terhadap kode atau fitur yang dikembangkan. Setelah semua masukan telah diterapkan dan sistem dinyatakan siap, proses dilanjutkan ke tahap finalisasi dan *deployment* ke lingkungan pengujian atau produksi. Sebagai bagian akhir dari proses, dilakukan penyusunan dokumentasi teknis untuk mendukung proses pemeliharaan

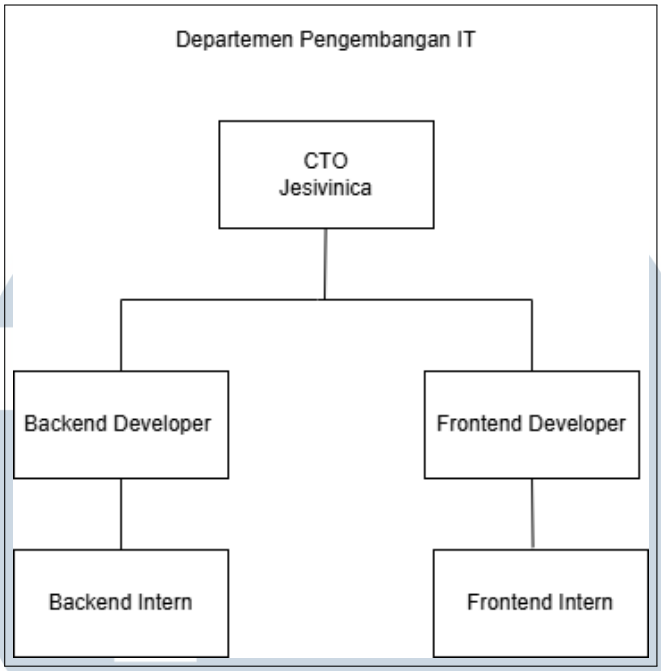
dan pengembangan lanjutan. Dengan selesainya dokumentasi tersebut, maka satu siklus kerja dapat dianggap selesai. Alur lengkap dari proses kerja tersebut digambarkan pada Gambar 3.1.



Gambar 3.1. Alur kerja *backend intern*

3.1.2 Alur Koordinasi *Backend Intern*

Sebagai *Backend Intern* di PT Jaya Santoso Teknologi, koordinasi dilakukan secara langsung dengan *Backend Developer* sebagai atasan teknis utama. Setiap tugas dan tanggung jawab yang diberikan oleh *Backend Developer* umumnya disampaikan melalui diskusi mingguan, baik secara daring maupun luring. *Backend Intern* bertugas untuk melaksanakan pekerjaan teknis seperti integrasi API, pembuatan *endpoint*, serta pengelolaan basis data sesuai arahan dan standar yang ditetapkan oleh *Backend Developer*. Selain itu, semua hasil kerja akan melalui proses *code review* untuk memastikan kualitas dan kesesuaian dengan kebutuhan sistem. *Backend Developer* juga berperan sebagai mentor yang memberikan bimbingan dalam menyelesaikan kendala teknis dan pengambilan keputusan selama implementasi. Koordinasi dilakukan secara intensif terutama saat pengerjaan fitur baru, *debugging error*, dan proses *deployment*. Dalam struktur organisasi yang ditunjukkan pada Gambar 3.2, *Backend Intern* berada langsung di bawah *Backend Developer* yang kemudian bertanggung jawab kepada CTO. Hubungan kerja yang terbentuk bersifat kolaboratif, memungkinkan *Backend Intern* untuk terus belajar dan berkembang dengan dukungan penuh dari tim pengembang inti.



Gambar 3.2. Alur koordinasi *backend intern*

3.2 Tugas yang Dilakukan

Selama masa magang di PT Jaya Santoso Teknologi, penulis bertanggung jawab dalam pengembangan sistem *Messaging Service* untuk mendukung kebutuhan komunikasi otomatis melalui berbagai kanal. Tiga tugas utama yang dilakukan selama kegiatan magang antara lain:

1. Bertugas membangun sistem pengiriman email massal yang digunakan untuk menyampaikan informasi penting secara serentak kepada banyak pengguna. Sistem ini mendukung pengiriman email dalam format teks maupun HTML, serta terintegrasi dengan vendor pihak ketiga seperti Mandrill dan Brevo untuk memastikan pengiriman berjalan lancar dan terverifikasi.

2. Mengembangkan fitur *Contact Us* yang memungkinkan pengguna menyampaikan pertanyaan, saran, atau keluhan melalui email. Fitur ini ditujukan untuk meningkatkan komunikasi dua arah antara pengguna dan administrator, dengan email dikirim secara otomatis menggunakan SMTP dan disimpan ke dalam sistem sebagai dokumentasi.

3. Membangun fitur pengiriman pesan WhatsApp yang digunakan untuk menyampaikan notifikasi atau informasi tertentu secara langsung kepada

pengguna melalui WhatsApp Business. Integrasi ini dilakukan menggunakan API resmi dari Meta, dengan pengujian dilakukan pada nomor yang telah diverifikasi selama proses aktivasi akun bisnis.

3.3 Uraian Pelaksanaan Magang

Selama kegiatan magang berlangsung, berbagai tugas yang dikerjakan dikelompokkan berdasarkan periode kerja selama 16 minggu. Setiap minggu memiliki fokus dan lingkup pekerjaan yang berbeda sesuai dengan kebutuhan proyek yang sedang berjalan. Rincian aktivitas serta tanggung jawab yang dilaksanakan selama periode magang dapat dilihat secara lebih jelas pada Tabel 3.1.

Tabel 3.1. Pekerjaan yang dilakukan setiap minggu

Minggu Ke -	Pekerjaan yang dilakukan
1-2	Melakukan perancangan awal sistem *backend* yang mencakup studi vendor layanan pesan, setup *environment* pengembangan menggunakan Spring Boot, perancangan skema basis data, dan pengujian awal API menggunakan Postman.
3-6	Implementasi pengiriman email menggunakan Mandrill, mencakup konfigurasi akun dan API, penyesuaian format *payload*, perbaikan *error handling*, *refactor* menggunakan *service layer*, serta *testing* pengiriman email.
7-11	Implementasi pengiriman email menggunakan Brevo, mencakup *testing* dan *debugging* pengiriman, perbaikan format *payload* dan skema database, implementasi *error handling*, *refactor* dengan DTO, serta penambahan fitur pengiriman HTML dan lampiran. Integrasi RabbitMQ dilakukan sebagai *middleware*, dan penyusunan dokumentasi dan koordinasi bersama tim *frontend*.
12-13	Implementasi dan *testing endpoint Contact Us*, termasuk pengalihan pengiriman email ke Gmail, validasi *input*, serta perapihan struktur kode dan *service*.
	Lanjut ke halaman berikutnya

Tabel 3.1. Pekerjaan yang dilakukan setiap minggu (lanjutan)

Minggu Ke -	Pekerjaan yang dilakukan
14	Implementasi *Swagger* untuk mendokumentasikan seluruh *endpoint* API pada sistem *Messaging Service*, sehingga memudahkan proses integrasi dan pemahaman teknis oleh tim pengembang lain. Selain itu, dilakukan diskusi dengan tim frontend untuk menyelaraskan struktur dokumentasi dan kebutuhan integrasi.
15-16	Implementasi WhatsApp dengan melakukan setup akun Meta Developer, mempelajari struktur API WhatsApp, pengujian pengiriman pesan menggunakan *token* sementara, *debugging* proses pengiriman, melengkapi data verifikasi, serta diskusi lanjutan dengan supervisor terkait tahapan integrasi WhatsApp Business ke dalam sistem.

3.3.1 Perancangan Awal Sistem *Backend*

Pada tahap awal pengembangan sistem *Messaging Service*, dilakukan serangkaian kegiatan perancangan yang menjadi fondasi dari implementasi *backend*. Kegiatan ini mencakup studi terhadap vendor layanan pesan untuk menentukan platform yang akan digunakan, penyiapan lingkungan pengembangan yang mendukung proses integrasi dan pengujian, serta perancangan skema basis data yang diperlukan untuk mencatat dan mengelola aktivitas pengiriman pesan.

A Studi Vendor

Dalam merancang sistem *Messaging Service*, tahap awal yang dilakukan adalah studi vendor layanan pengiriman email untuk menentukan solusi yang paling sesuai dengan kebutuhan integrasi. Beberapa vendor yang dianalisis antara lain Mandrill, Brevo, Amazon SES, SendGrid, Mailchimp, dan Mailersend. Penilaian dilakukan berdasarkan dokumentasi resmi, struktur *payload*, kemudahan autentikasi, dukungan format HTML, kecepatan pengiriman, dan model biaya. Hasil analisis tersebut disajikan dalam Tabel 3.2 sebagai acuan dalam pengambilan keputusan. Dari studi ini, Mandrill dipilih untuk pengiriman email transaksional, sementara Brevo digunakan untuk kebutuhan pengiriman email massal dengan

dukungan antrian pesan.

Tabel 3.2. Perbandingan vendor layanan pengiriman email

Layanan	**Kelebihan**	**Kekurangan**	**Cocok untuk**
Mailchimp	Antarmuka intuitif, fitur otomatisasi lengkap, integrasi dengan berbagai *platform*, analitik mendalam	Branding Mailchimp pada paket gratis, harga relatif tinggi untuk fitur lanjutan, wajib email domain bisnis	*Startup* dan bisnis kecil-menengah dengan kebutuhan marketing digital
Brevo	Dukungan email dan SMS, otomatisasi kuat, tanpa batas total bulanan	UI kurang intuitif, batas pengiriman harian di paket gratis, wajib email domain bisnis	*E-commerce* dan bisnis menengah dengan kebutuhan *multi-channel*
Amazon SES	Harga sangat murah, cocok untuk skala besar, integrasi dengan layanan AWS lainnya	Setup awal kompleks, tidak memiliki UI visual, butuh kemampuan teknis tinggi	*Enterprise* dan sistem internal berskala besar
Mandrill (Mailchimp)	Kuat untuk pengiriman transaksional email, integrasi dengan Mailchimp, laporan analitik rinci	Wajib akun Mailchimp berbayar, harga tinggi, integrasi kompleks	Aplikasi dan web yang mengirim email transaksional

Lanjut ke halaman berikutnya

Tabel 3.2 Perbandingan vendor layanan pengiriman email (lanjutan)

Layanan	Kelebihan	Kekurangan	Cocok untuk
SendGrid	Dokumentasi lengkap, dukung templating dan statistik, antarmuka baik	Sering masuk spam jika domain belum tervalidasi, rate limit di paket gratis	Platform SaaS, dan notifikasi sistem otomatis
Mailersend	Tampilan modern, integrasi mudah, mendukung templating dan analytics	Verifikasi domain manual, fitur lanjutan terbatas untuk skala bisnis kecil-menengah	pemilik bisnis kecil-menengah

Selain itu, layanan Gmail juga digunakan untuk pengiriman pesan melalui fitur *Contact Us* pada platform. Gmail dipilih karena bersifat gratis, memiliki dukungan SMTP yang stabil, serta dapat langsung diintegrasikan dengan pustaka email berbasis Java di Spring Boot. Meskipun bukan ditujukan untuk skala besar, Gmail cukup andal untuk menangani pengiriman pesan dari pengguna kepada admin, terutama karena volume pesan dari fitur ini relatif rendah.

Untuk layanan pengiriman pesan melalui WhatsApp, studi dilakukan terhadap dua penyedia yaitu Qontak dan Meta Developer. Meskipun Qontak menyediakan fitur antarmuka siap pakai dan kemudahan integrasi, biaya layanan Qontak tergolong tinggi dan kurang cocok untuk sistem yang sedang dibangun dari awal dengan kebutuhan fleksibilitas tinggi. Oleh karena itu, sistem memilih pendekatan menggunakan Meta Developer karena memberikan fleksibilitas teknis yang lebih besar, kontrol penuh terhadap proses pengiriman pesan, serta efisiensi biaya yang lebih baik dalam jangka panjang.

B Penyiapan Lingkungan Pengembangan

Penyiapan lingkungan pengembangan merupakan tahap awal yang sangat penting untuk memastikan seluruh proses implementasi backend berjalan stabil dan konsisten. Dalam proyek Messaging Service, secara eksplisit software dan aplikasi yang digunakan adalah sebagai berikut:

1. **Software Pendukung**

   (a) Java Development Kit (JDK) 17

   Digunakan sebagai runtime utama dalam proyek ini karena JDK 17 merupakan versi Long Term Support (LTS) yang direkomendasikan untuk pengembangan backend modern, termasuk mendukung penuh Spring Boot 3.4.2 [4]. Selain stabil, JDK 17 juga menyediakan fitur-fitur baru seperti enhanced switch expressions, sealed classes, dan peningkatan performa garbage collector yang mendukung efisiensi memori pada aplikasi berskala besar. Dengan menggunakan JDK 17, pengembang dapat memastikan kompatibilitas jangka panjang serta keamanan saat melakukan integrasi layanan eksternal seperti database dan message broker.

   (b) Spring Boot 3.4.2

   Framework backend berbasis Java yang digunakan dalam proyek ini adalah Spring Boot versi 3.4.2. Versi ini dipilih karena menyediakan pembaruan keamanan, stabilitas, dan fitur modern seperti konfigurasi berbasis anotasi yang lebih ringkas, integrasi native ke container (misalnya Docker), serta peningkatan pada Spring Data, Spring Security, dan Spring Web. Spring Boot 3.4.2 mempermudah pembuatan RESTful API, manajemen dependensi melalui Spring Initializr, serta memfasilitasi integrasi dengan berbagai layanan eksternal seperti email service (Brevo dan Mandrill), message broker (RabbitMQ), dan database (PostgreSQL). Dengan kemampuan auto-configuration yang disediakan, pengembang dapat fokus pada logika bisnis tanpa harus menulis konfigurasi manual yang kompleks [5].

   (c) Docker

   Digunakan untuk menjalankan RabbitMQ dalam lingkungan yang terisolasi, konsisten, dan mudah dikonfigurasi [6].

   (d) RabbitMQ

   Digunakan sebagai message broker (*middleware*) untuk menangani antrean pengiriman email massal agar lebih stabil dan tidak bergantung pada respons langsung server pihak ketiga.

2. **Aplikasi Pendukung**

   (a) IntelliJ IDEA Community Edition

Integrated Development Environment (IDE) yang digunakan untuk menulis kode Java, menjalankan server lokal, debugging, serta mengelola struktur paket Spring Boot [7].

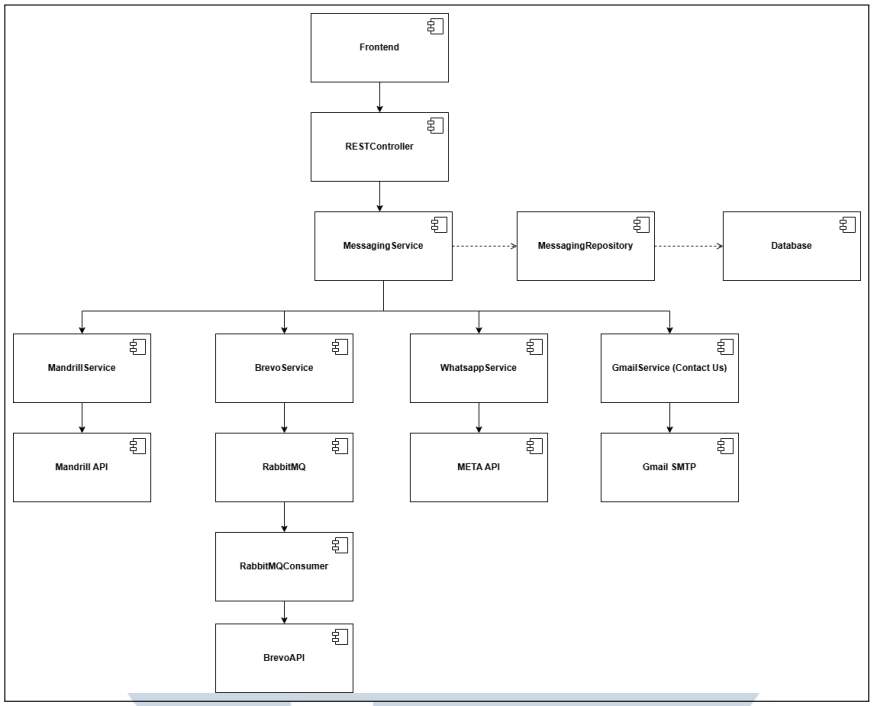
(b) pgAdmin
Aplikasi antarmuka grafis untuk mengelola database PostgreSQL, termasuk pembuatan tabel, memeriksa data, serta memonitor performa basis data [8].

(c) Postman
Digunakan sebagai alat pengujian API, memvalidasi payload, mengirim permintaan HTTP, serta memverifikasi respons server secara langsung [9].

(d) GitHub
Platform kolaborasi pengembangan kode, kontrol versi, serta integrasi antar modul dalam satu repositori tim [10].

C Perancangan Arsitektur Komponen Sistem

Pada tahap awal pengembangan, dilakukan perancangan arsitektur komponen untuk menggambarkan interaksi antar modul utama dalam sistem *Messaging Service*. Diagram komponen ditunjukkan pada Gambar 3.3, yang merepresentasikan alur komunikasi dari frontend hingga berbagai layanan pengiriman pesan eksternal.



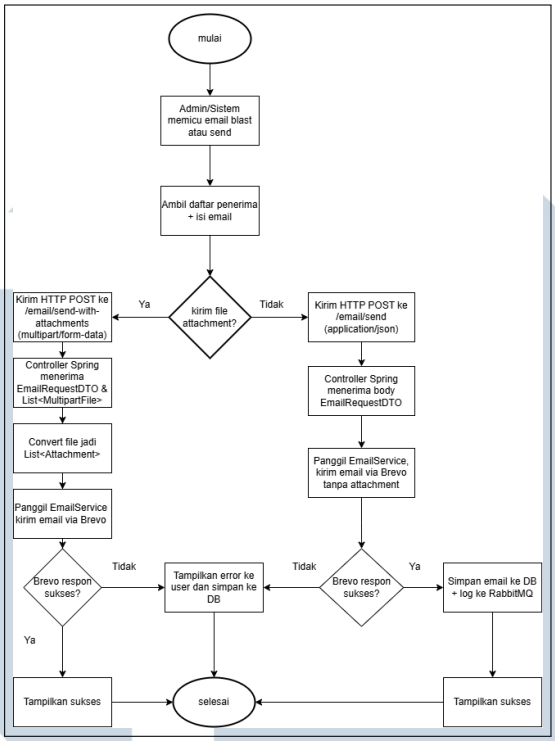
Gambar 3.3. Diagram komponen sistem *messaging service*

Sistem dimulai dari *Frontend* yang mengirimkan permintaan pengiriman pesan ke *RESTController*. Komponen ini bertugas menerima permintaan HTTP, memvalidasi parameter, lalu meneruskannya ke *MessagingService* sebagai pusat logika bisnis. Selanjutnya, *MessagingService* menentukan layanan pengiriman yang digunakan sesuai jenis pesan, seperti Mandrill, Brevo, WhatsApp, atau Gmail untuk fitur *Contact Us*. Setiap layanan memiliki service tersendiri, yaitu *MandrillService*, *BrevoService*, *WhatsappService*, dan *GmailService*. Untuk pengiriman email melalui Mandrill, pesan dikirim langsung ke *Mandrill API*. Sementara itu, untuk pengiriman menggunakan Brevo, sistem menerapkan pendekatan asinkron dengan perantara RabbitMQ. Pesan dari *BrevoService* dikirim ke antrean RabbitMQ, kemudian dikonsumsi oleh *RabbitMQConsumer* sebelum diteruskan ke *Brevo API*. Layanan WhatsApp diintegrasikan menggunakan *META API*, sedangkan pengiriman pesan *Contact Us* dilakukan melalui protokol *SMTP* menggunakan akun Gmail. Selain itu, semua aktivitas pengiriman pesan dicatat melalui *MessagingRepository*, yang berkomunikasi dengan *Database* untuk menyimpan histori pengiriman.

D Perancangan Alur Proses Pengiriman Pesan

Perancangan alur proses pengiriman pesan bertujuan untuk menggambarkan bagaimana sistem memproses permintaan pengiriman pesan dari pengguna, meneruskannya ke layanan eksternal seperti Brevo, Gmail SMTP, Meta WhatsApp API, atau Mandrill, serta menyimpan data pesan ke dalam basis data. Proses ini divisualisasikan menggunakan *flowchart*, karena lebih sesuai untuk menunjukkan alur logika sistem yang tidak berorientasi objek, serta mempermudah dalam menjelaskan pengambilan keputusan dan percabangan proses di setiap langkah [11].

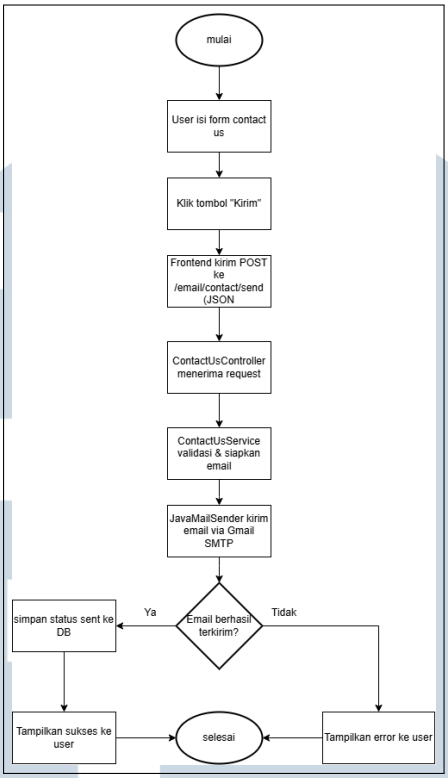
1. Pengiriman Email via Brevo
   Gambar 3.4 memperlihatkan *flowchart* alur pengiriman email menggunakan layanan Brevo. Proses dimulai dari admin yang mengisi formulir pengiriman email melalui antarmuka *frontend*. Sistem akan melakukan validasi terhadap input, termasuk mengecek apakah ada lampiran yang disertakan. Jika input valid, sistem menyimpan data email ke dalam database dengan status awal *PENDING*, kemudian meneruskan data ke antrean pesan (RabbitMQ). Selanjutnya, sistem memanggil API Brevo untuk mengirim email. Setelah Brevo merespons, sistem memperbarui status email di database menjadi *SENT* atau *FAILED*. Dengan demikian, setiap status email dapat dilacak dan diupdate sesuai hasil pengiriman.



Gambar 3.4. *Flowchart* pengiriman email via Brevo

2. Pengiriman Pesan melalui Fitur *Contact Us*

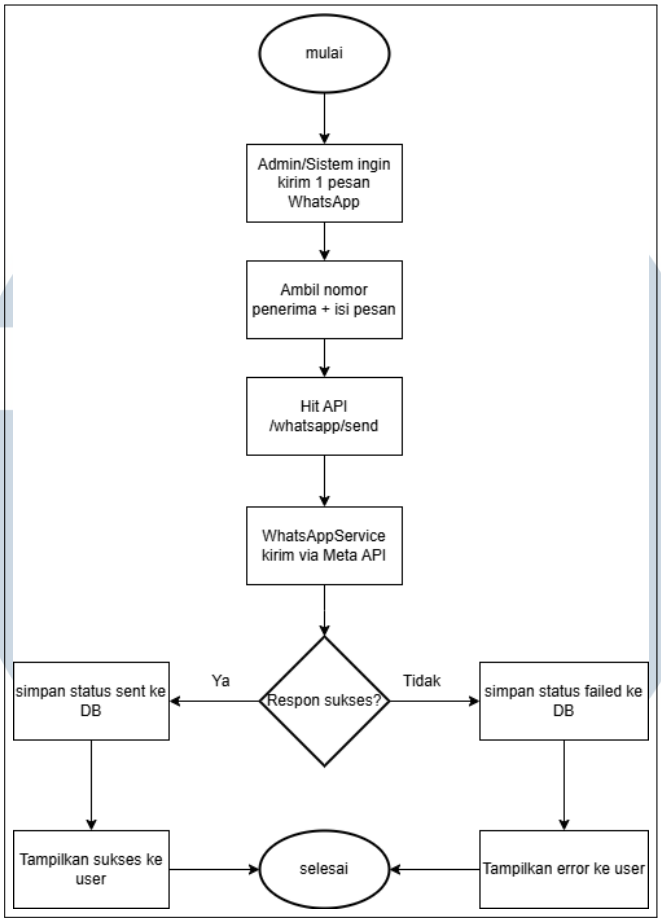
   Gambar 3.5 menunjukkan *flowchart* alur proses ketika pengguna mengirim pesan melalui fitur *Contact Us*. Setelah pengguna mengisi form, sistem akan melakukan validasi data seperti nama, email, dan pesan. Jika valid, sistem langsung memanggil SMTP Gmail untuk mengirim email ke admin. Apabila pengiriman berhasil, data pesan disimpan ke database untuk kebutuhan arsip dan audit. Namun, jika gagal, sistem menampilkan notifikasi kegagalan kepada pengguna, dan data tidak disimpan. Hal ini dirancang agar hanya pesan yang benar-benar terkirim yang tercatat.



Gambar 3.5. *Flowchart* proses *Contact Us* via Gmail SMTP

3. Pengiriman Pesan WhatsApp via Meta API

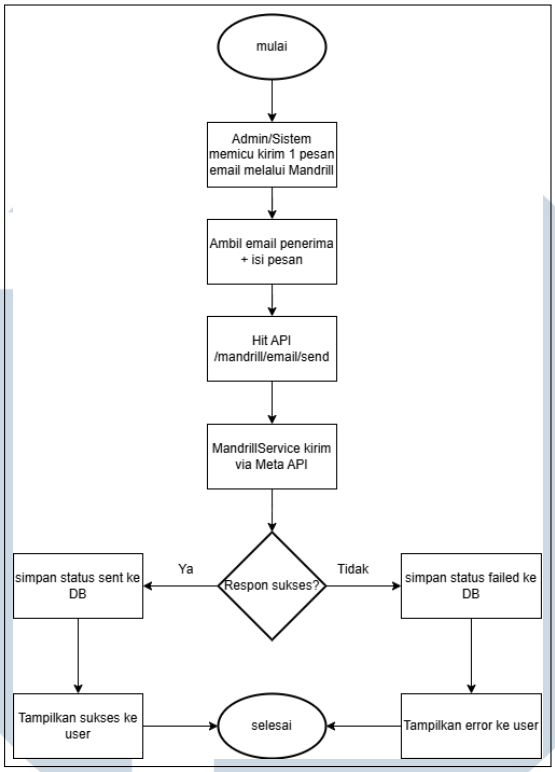
   Gambar 3.6 memperlihatkan alur pengiriman pesan WhatsApp melalui layanan Meta API. Setelah admin mengisi pesan dan nomor tujuan di antarmuka, sistem memvalidasi data input. Jika valid, sistem memanggil Meta API untuk mengirim pesan WhatsApp. Respons dari Meta API, baik sukses maupun gagal, selalu dicatat ke database untuk kepentingan audit dan pelacakan. Dengan pencatatan ini, admin dapat mengetahui status pesan secara detail, termasuk jika terjadi kesalahan seperti token kadaluwarsa atau nomor tidak valid.



Gambar 3.6. *Flowchart* pengiriman WhatsApp via Meta API

4. Pengiriman Email via Mandrill

   Gambar 3.7 menggambarkan *flowchart* alur pengiriman email melalui Mandrill. Admin mengirim permintaan melalui antarmuka aplikasi. Sistem memeriksa kelengkapan dan validitas data. Jika valid, sistem langsung memanggil layanan eksternal Mandrill untuk melakukan pengiriman email. Setelah menerima respons, sistem akan mencatat data ke database dengan status *SENT* atau *FAILED*. Status ini akan ditampilkan kepada admin sehingga admin bisa memantau hasil pengiriman secara real-time. Pendekatan ini juga membantu menjaga akurasi data log, terutama jika digunakan untuk analisis performa pengiriman.



Gambar 3.7. *Flowchart* pengiriman email via Mandrill

E Struktur Tabel dan Relasi Basis Data

Basis data adalah sekumpulan data yang terorganisir secara sistematis dan dapat diakses, dikelola, serta diperbarui dengan mudah menggunakan sistem manajemen basis data[12]. Dalam konteks sistem informasi, basis data berfungsi sebagai komponen utama untuk menyimpan informasi secara permanen, menjaga integritas data, serta mendukung efisiensi proses pencarian dan pelaporan data Perancangan struktur basis data dalam sistem *Messaging Service* disesuaikan dengan kebutuhan masing-masing layanan, yaitu Mandrill, Brevo, WhatsApp, dan fitur *Contact Us*. Tujuan utama dari perancangan ini adalah untuk memastikan bahwa sistem dapat mencatat aktivitas pengiriman pesan secara konsisten dan efisien. Struktur basis data yang tepat juga berfungsi sebagai landasan dalam proses *logging*, pemantauan performa, serta integrasi dengan *middleware* seperti RabbitMQ . Secara umum, sistem dibagi menjadi layanan yang mencatat pesan dalam bentuk email maupun pesan berbasis nomor telepon. Brevo sebagai vendor utama memiliki struktur yang paling kompleks dengan dua tabel yang saling berelasi, sedangkan layanan lain seperti Mandrill, WhatsApp, dan *Contact Us* menggunakan struktur tabel tunggal. Berikut ini adalah uraian struktur basis data

berdasarkan masing-masing layanan:

- Layanan Mandrill

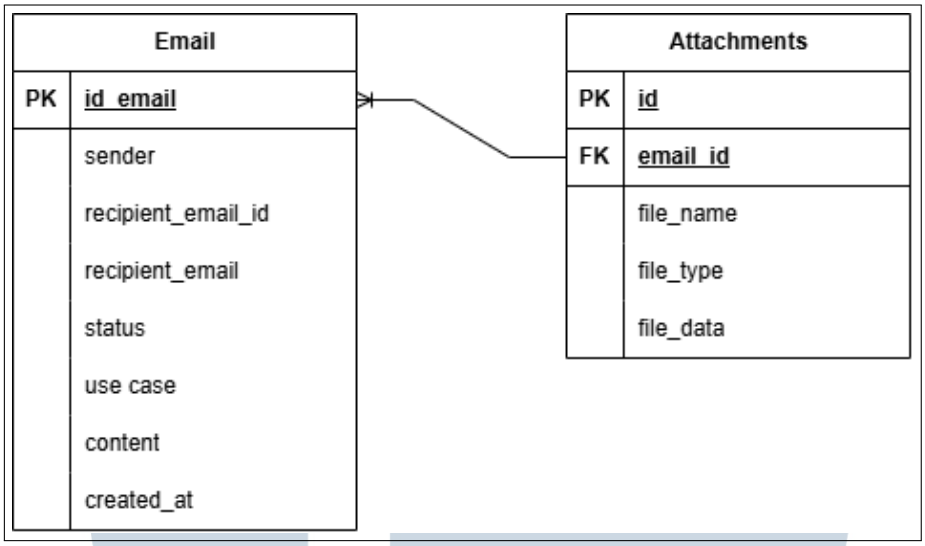
  Layanan Mandrill hanya menggunakan satu tabel untuk mencatat aktivitas pengiriman email. Tabel 3.3 menyimpan data email yang dikirim melalui layanan Mandrill. Setiap entri mencakup pengirim, penerima, status pengiriman, hingga isi konten email.

Tabel 3.3. Struktur tabel messages untuk layanan Mandrill

Kolom	**Tipe data**	**Keterangan**
id_email	SERIAL	Primary key
sender	VARCHAR(255)	Alamat email pengirim
recipient_email_id	VARCHAR(255)	ID penerima (jika ada)
recipient_email	VARCHAR(255)	Alamat email tujuan
status	VARCHAR(50)	Status pengiriman
use_case	VARCHAR(100)	Jenis penggunaan
content	TEXT	Isi pesan
created_at	TIMESTAMP	Waktu pembuatan pesan

- Layanan Brevo

  Brevo memiliki dua tabel utama yaitu *email* dan *attachments* yang saling terhubung melalui relasi one-to-many, di mana satu email dapat memiliki banyak lampiran. Gambar 3.8 menunjukkan bahwa setiap entri email dapat memiliki banyak data lampiran (*attachments*) yang terkait berdasarkan foreign key *email_id*. Struktur ini memungkinkan pencatatan yang fleksibel untuk pengiriman pesan beserta file pendukungnya.



Gambar 3.8. *Entity relationship diagram* untuk layanan Brevo

Tabel 3.4 menyimpan informasi terkait file yang dikirim sebagai lampiran pada email. Setiap lampiran dikaitkan dengan satu email melalui kolom *email_id*.

Tabel 3.4. Struktur tabel attachments untuk lampiran pesan

Kolom	**Tipe data**	**Keterangan**
id	SERIAL	Primary key
email_id	INTEGER	Foreign key ke *id_email* di tabel *email*
file_name	VARCHAR(255)	Nama file lampiran
file_type	VARCHAR(100)	Tipe MIME file
file_data	BYTEA	Data file dalam bentuk biner

Tabel 3.5 berisi informasi inti dari pengiriman pesan menggunakan Brevo, seperti pengirim, penerima, konten, dan waktu pengiriman.

Tabel 3.5. Struktur tabel email untuk layanan pengiriman pesan

Kolom	Tipe data	Keterangan
id_email	SERIAL	Primary key
sender	VARCHAR(255)	Alamat email pengirim
recipient_email_id	VARCHAR(255)	ID penerima (opsional)
recipient_email	VARCHAR(255)	Alamat email tujuan
status	VARCHAR(50)	Status pengiriman pesan
use_case	VARCHAR(100)	Jenis penggunaan pesan
content	TEXT	Isi konten email
created_at	TIMESTAMP	Waktu pembuatan pesan

- Layanan Contact Us Email

  Fitur *Contact Us* menggunakan struktur tabel sederhana yang berisi informasi dasar dari pengunjung yang mengirimkan pesan, seperti nama, email, subjek, dan isi pesan. Data ini disimpan langsung ke dalam satu tabel tanpa relasi tambahan karena tidak membutuhkan lampiran atau histori pengiriman yang kompleks. Fitur ini memungkinkan pengguna mengirimkan pertanyaan atau pesan ke admin. Data pesan dicatat dalam Tabel 3.6.

Tabel 3.6. Struktur tabel contact_us_message_email

Kolom	Tipe data	Keterangan
id	SERIAL	Primary key
first_name	VARCHAR(100)	Nama depan pengirim
last_name	VARCHAR(100)	Nama belakang pengirim
email	VARCHAR(150)	Alamat email pengirim
message	TEXT	Isi pesan
created_at	TIMESTAMP	Waktu pesan dikirimkan

- Layanan WhatsApp

  Untuk layanan WhatsApp, perancangan awal dilakukan dengan mempersiapkan struktur tabel yang fleksibel untuk menyimpan nomor tujuan, isi pesan, dan status pengiriman. Tabel 3.7 menyimpan data pengiriman pesan melalui platform WhatsApp yang dikelola melalui Meta Developer.

Tabel 3.7. Struktur tabel whatsapp_messages

Kolom	Tipe data	Keterangan
id	SERIAL	Primary key
recipient_number	VARCHAR(20)	Nomor penerima
message_type	VARCHAR(20)	Jenis pesan (text/media)
message_body	TEXT	Isi pesan teks
media_id	VARCHAR(100)	ID media
media_link	TEXT	Tautan media
media_filename	VARCHAR(255)	Nama file media
caption	TEXT	Keterangan media
preview_url	BOOLEAN	Tampilkan pratinjau URL
status	VARCHAR(20)	Status pengiriman
api_response	TEXT	Respons dari API
created_at	TIMESTAMP	Waktu pembuatan pesan
updated_at	TIMESTAMP	Waktu pembaruan pesan

3.3.2 Implementasi Layanan Mandrill

Pada tahap ini, sistem *Messaging Service* diimplementasikan dengan menggunakan layanan email dari Mandrill sebagai vendor eksternal. Tahapan implementasi mencakup konfigurasi API, pembuatan struktur *payload*, penanganan kesalahan, serta *refactor* menggunakan layer layanan. Pengujian terhadap *endpoint* Mandrill juga dilakukan untuk memastikan seluruh alur berjalan sesuai fungsinya. Kode 3.1 menunjukkan konfigurasi awal yang digunakan untuk menyimpan informasi kredensial dan endpoint dari layanan pihak ketiga.

```
# Mandrill API configuration
vendor.email.mandrill.api.url=${VENDOR_EMAIL_MANDRILL_API_URL}
vendor.email.mandrill.api.key=${VENDOR_EMAIL_MANDRILL_API_KEY}
```

Kode 3.1: Konfigurasi Mandrill di file *application.properties*

A Pengambilan Konfigurasi API Mandrill

Tahap awal implementasi dimulai dengan konfigurasi akun Mandrill, mengambil konfigurasi *application.properties* dengan teknik *dependency injection* seperti yang terlihat pada Kode 3.2. Kode tersebut memperlihatkan konstruktor

dari kelas *MandrillService* yang menggunakan objek *RestTemplate* sebagai klien *HTTP* untuk melakukan komunikasi dengan *API* eksternal, dalam hal ini layanan Mandrill. Nilai-nilai seperti *mandrillApiKey* dan *mandrillUrl* disuntikkan ke dalam konstruktor menggunakan anotasi *@Value*, yang mengambil nilainya dari berkas konfigurasi *application.properties*. Dengan pola ini, informasi sensitif seperti *API key* tidak ditulis langsung di dalam kode sumber, melainkan disimpan terpisah untuk alasan keamanan dan kemudahan pengelolaan. Selanjutnya, *RestTemplate* akan menggunakan *URL* dan *API key* ini untuk mengirim permintaan *HTTP POST* ke endpoint Mandrill untuk mengirim email.

```
1 public MandrillService(RestTemplate restTemplate,
2                         @Value("${vendor.email.mandrill.api.key
    }") String mandrillApiKey,
3                         @Value("${vendor.email.mandrill.api.url
    }") String mandrillUrl) {
4         this.restTemplate = restTemplate;
5         this.mandrillApiKey = mandrillApiKey;
6         this.mandrillUrl = mandrillUrl;
7     }
```

Kode 3.2: Konfigurasi Mandrill di file *MandrillService*

B Struktur *Payload* dan Format Permintaan

Permintaan ke Mandrill dibuat dalam bentuk *payload* berformat JSON, seperti ditunjukkan pada Kode 3.3. Format ini berisi informasi email seperti alamat pengirim, penerima, subjek, dan konten email dalam bentuk HTML maupun teks biasa. Struktur *payload* ini merupakan representasi data internal yang dikonstruksi oleh sistem sebelum dikonversi ke dalam format yang sesuai dengan standar API Mandrill. Nilai-nilai ini nantinya akan disisipkan ke dalam *body* permintaan HTTP POST untuk diproses dan dikirim oleh layanan Mandrill.

```
1 {
2   "recipientEmail": "tamu1@example.com",
3   "sender": "wedding@minyma.id",
4   "useCase": "Undangan Pernikahan",
5   "content": "Dengan hormat, kami mengundang Anda untuk hadir di
     acara pernikahan kami."
6 }
```

Kode 3.3: Struktur *payload* pengiriman email melalui Mandrill

C Struktur Komponen dan Refactor *Service Layer*

Awalnya, logika bisnis untuk pengiriman email ditulis langsung di dalam kelas *Controller*, yang menyebabkan kode menjadi tidak modular dan sulit untuk diuji secara terpisah. Refaktorisasi dilakukan dengan memisahkan tanggung jawab ke dalam beberapa *layer*, yaitu *Controller*, *Service*, dan *Repository*. Pemisahan ini mengikuti prinsip *Service Layer Pattern* yang umum diterapkan dalam arsitektur Spring Boot.

Sebelum dilakukan refaktorisasi, pengiriman email dilakukan secara langsung di dalam *Controller*, termasuk pembuatan *payload*, pemanggilan API, dan penyimpanan ke basis data, sebagaimana ditunjukkan pada Kode 3.4.

```
@PostMapping("/send")
public ResponseEntity<String> sendEmail(@RequestBody Message
    message) {
    // Buat payload dan kirim ke Mandrill
    // Simpan ke database
    // Return response
}
```

Kode 3.4: Kode pengiriman email sebelum refaktor

Setelah refaktorisasi, logika pengiriman dipindahkan ke kelas *MessageService*, sementara pemanggilan API eksternal ditangani oleh *MandrillService*. Kelas *Controller* kini hanya bertugas menerima dan meneruskan *request* dari pengguna, sebagaimana diperlihatkan pada Kode 3.5.

```
@PostMapping("/send")
public ResponseEntity<String> sendMessage(@RequestBody Message
    messageRequest) {
    return messageService.sendMessage(messageRequest);
}
```

Kode 3.5: Kode setelah refaktor dengan pemisahan *service*

Berikut merupakan pembagian tanggung jawab setiap komponen setelah proses refaktorisasi:

- *Controller*: menerima *HTTP request* dari pengguna.
- *MessageService*: memvalidasi, mengatur logika penyimpanan dan delegasi pengiriman.
- *MandrillService*: membentuk *payload* dan memanggil API eksternal menggunakan *RestTemplate*.

- *Repository*: menyimpan data ke basis data jika berhasil.

D Logika Pengiriman Email ke Mandrill

Pseudocode proses pengiriman email ditunjukkan pada Kode 3.6. Prosedur ini merepresentasikan logika pengiriman email menggunakan layanan Mandrill secara terstruktur, mulai dari pengaturan data, pemanggilan API, hingga penyimpanan ke basis data.

```
PROCEDURE SendMandrillEmail(requestData)

    IF requestData.sender IS EMPTY THEN
        sender TO "defaultsender@minyma.id"
    ELSE
        sender TO requestData.sender
    END IF

    recipient = requestData.recipientEmail
    subject = "Subject: " + requestData.useCase
    content = requestData.content

    response = CallMandrillAPI(sender, recipient, subject, content
    )

    IF response.status == 200 THEN
        message = NEW Message
        message.sender = sender
        message.recipientEmailId = requestData.recipientEmailId
        message.recipientEmail = recipient
        message.status = "SENT"
        message.useCase = requestData.useCase
        message.content = content
        message.createdAt = CURRENT_TIMESTAMP

        SAVE message TO database

        RETURN "Email sent & saved to database."
    ELSE
        RETURN "Failed to send email."
    END IF

```

```
END PROCEDURE
```

Kode 3.6: Pseudocode prosedural pengiriman email melalui Mandrill

Prosedur diawali dengan pemeriksaan nilai *sender*. Jika kosong, maka sistem menggunakan alamat pengirim default *defaultsender@minyma.id*. Selanjutnya, data seperti penerima, subjek, dan isi email diambil dari *requestData*, kemudian dikirim ke API Mandrill menggunakan fungsi *CallMandrillAPI*. Jika status balikan menyatakan berhasil (*status == 200*), maka data email disimpan ke dalam basis data melalui entitas *Message*, lalu mengembalikan pesan sukses. Jika gagal, prosedur mengembalikan pesan *error*.

Dengan pendekatan ini, logika pengiriman email bersifat modular, mudah diuji, dan tetap menjaga pencatatan ke basis data ketika pengiriman berhasil.

E *Error Handling* dan Validasi

Jika terjadi kesalahan, seperti *API key* yang tidak valid atau kegagalan jaringan, sistem akan mengembalikan kode status HTTP *500 Internal Server Error* disertai informasi *error*, sebagaimana ditunjukkan pada Kode 3.7.

```
{
  "error": "Failed to send email",
  "details": "401 Unauthorized"
}
```

Kode 3.7: Contoh respons kegagalan pengiriman email

Penanganan kesalahan dilakukan menggunakan blok *try-catch* di dalam kelas *MessageService*, sehingga kesalahan dalam pemanggilan *API* tidak menyebabkan gangguan pada kestabilan sistem secara keseluruhan. Jika pengiriman gagal, sistem akan mengembalikan status sesuai balasan dari layanan *Mandrill*, atau *500* jika terjadi pengecualian di sisi server.

Logika pengiriman email serta mekanisme *error handling* disusun dalam bentuk *pseudocode* seperti ditunjukkan pada Listing 3.8.

```
FUNCTION sendMessage(messageRequest)
    IF messageRequest.sender IS NULL OR EMPTY THEN
        SET senderEmail TO "defaultsender@minyma.id"
    ELSE
        SET senderEmail TO messageRequest.sender
    ENDIF

```

```
    TRY
        CALL MandrillService.sendEmail() -> response

        IF response.status IS 2xx THEN
            CREATE message OBJECT
            CALL messageRepository.save(message)
            RETURN HTTP 200 OK WITH MESSAGE "Email sent & saved to database."
        ELSE
            RETURN response.status WITH MESSAGE "Failed to send email."
        ENDIF

    CATCH Exception e
        RETURN HTTP 500 INTERNAL SERVER ERROR WITH MESSAGE e.message
    ENDTRY
END FUNCTION
```

Kode 3.8: Pseudocode proses pengiriman email dan *error handling*

F Pengujian *Endpoint* Mandrill

Pengujian dilakukan untuk memastikan bahwa setiap *endpoint* pada layanan Mandrill dapat berjalan sesuai fungsinya. Setiap *endpoint* diuji menggunakan Postman, mulai dari proses pengiriman, pembacaan, pembaruan, hingga penghapusan data email. Tabel 3.8 merangkum daftar *endpoint* yang diuji beserta fungsinya.

Tabel 3.8. Daftar *endpoint* Mandrill dan deskripsinya

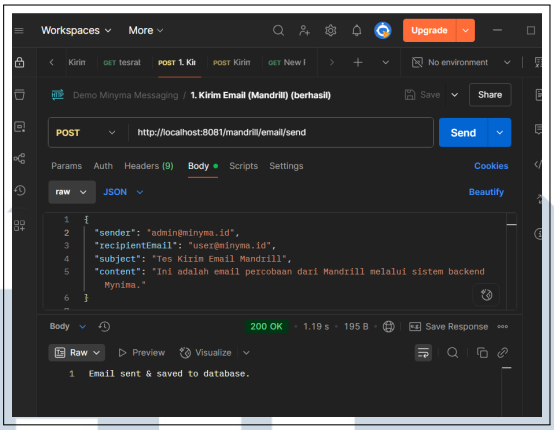
No	Nama Endpoint	Method dan URL	Deskripsi
1	Kirim Email (Mandrill)	POST /mandrill/email/send	Mengirim email menggunakan vendor Mandrill. Endpoint ini tidak dilanjutkan karena perubahan kebijakan layanan.
Lanjut ke halaman berikutnya			

Tabel 3.8 daftar *endpoint* Mandrill dan deskripsinya (lanjutan)

No	Nama Endpoint	Method dan URL	Deskripsi
2	Ambil Email by ID (Mandrill)	GET /mandrill/email/{idEmail}	Mendapatkan data email dari sistem yang terhubung ke Mandrill berdasarkan ID.
3	Update Email (Mandrill)	PUT /mandrill/email/{idEmail}	Memperbarui isi email dari database untuk vendor Mandrill berdasarkan ID. Tidak mengubah konten di sistem Mandrill.
4	Hapus Email (Mandrill)	DELETE /mandrill/email/{idEmail}	Menghapus data email dari basis data lokal berdasarkan ID.

Endpoint *GET*, *PUT*, dan *DELETE* berfungsi untuk mengambil, memperbarui, atau menghapus data email dari basis data lokal. Tidak terdapat pemanggilan ulang ke vendor *Mandrill* pada proses tersebut. Oleh karena itu, dokumentasi teknis secara mendalam hanya difokuskan pada endpoint *POST* untuk pengiriman email yang mencakup integrasi ke vendor eksternal. Meski begitu, pengujian tetap dilakukan terhadap seluruh *endpoint* menggunakan *Postman*. Respons sukses maupun *error* dievaluasi untuk memastikan bahwa setiap endpoint berfungsi sesuai spesifikasi.

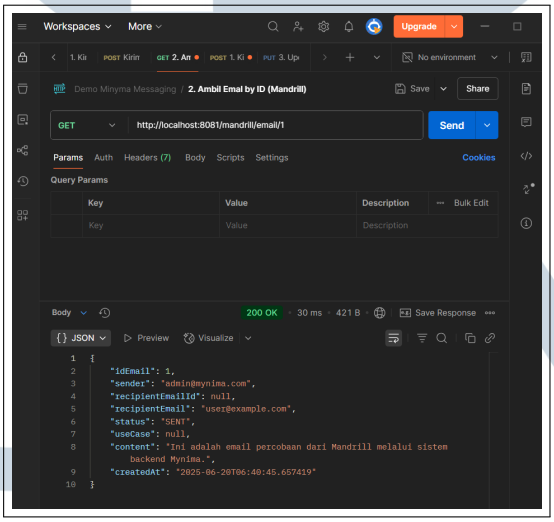
1. Pengujian *endpoint* pengiriman email
   Endpoint POST /mandrill/email/send digunakan untuk mengirim email melalui vendor Mandrill. Meskipun tidak digunakan secara aktif, *endpoint* ini tetap diuji dan terlihat pada Gambar 3.9 berhasil mengembalikan respons sukses.



Gambar 3.9. Pengujian *endpoint* pengiriman email melalui Mandrill

2. Pengujian *endpoint* pengambilan email berdasarkan ID

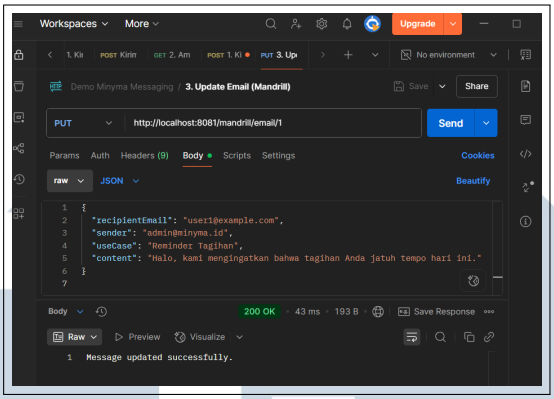
   Endpoint GET /mandrill/email/{idEmail} berfungsi untuk mengambil data email berdasarkan ID. Gambar 3.10 menunjukkan bahwa sistem dapat menampilkan data email sesuai permintaan.



Gambar 3.10. Pengujian *endpoint* ambil data email berdasarkan ID

3. Pengujian *endpoint* pembaruan email

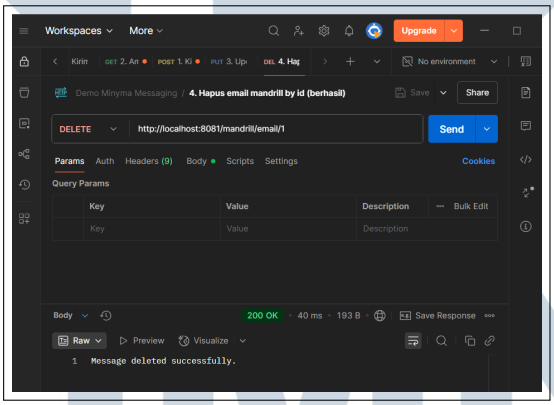
   Endpoint PUT /mandrill/email/{idEmail} digunakan untuk memperbarui isi email di basis data berdasarkan ID. Hasil pengujian terlihat pada Gambar 3.11 yang menunjukkan bahwa perubahan berhasil disimpan secara lokal.



Gambar 3.11. Pengujian *endpoint* pembaruan data isi email berdasarkan ID

4. Pengujian *endpoint* penghapusan email

   Endpoint DELETE /mandrill/email/{idEmail} berfungsi untuk menghapus data email dari basis data. Gambar 3.12 menunjukkan bahwa data berhasil dihapus dan sistem merespons dengan sukses.



Gambar 3.12. Pengujian *endpoint* penghapusan email berdasarkan ID

3.3.3 Implementasi Brevo dan RabbitMQ

Sistem pengiriman email yang dibangun menggunakan integrasi ke vendor Brevo serta *middleware* RabbitMQ bertujuan untuk mengoptimalkan penanganan beban tinggi dan mendukung pengiriman pesan secara *asinkron*. Integrasi RabbitMQ memungkinkan proses pengiriman email tidak dilakukan secara langsung oleh *controller*, tetapi didorong melalui *message queue*, sehingga meningkatkan skalabilitas dan ketahanan sistem saat menangani permintaan secara paralel. Kode 3.9 menunjukkan konfigurasi awal yang digunakan untuk menyimpan kredensial dan parameter teknis yang berkaitan dengan layanan pihak ketiga.

Konfigurasi ini disimpan di dalam berkas *application.properties* dan menggunakan format *environment variable* agar dapat disesuaikan pada berbagai lingkungan (*development*, *staging*, atau *production*).

```
# Brevo API configuration
vendor.email.brevo.api.url=${VENDOR_EMAIL_BREVO_API_URL}
vendor.email.brevo.api.key=${VENDOR_EMAIL_BREVO_API_KEY}

# RabbitMQ configuration
spring.rabbitmq.host=${RABBITMQ_HOST}
spring.rabbitmq.port=${RABBITMQ_PORT}
spring.rabbitmq.username=${RABBITMQ_USERNAME}
spring.rabbitmq.password=${RABBITMQ_PASSWORD}
rabbitmq.queue.name=${RABBITMQ_QUEUE_NAME}
rabbitmq.exchange.name=${RABBITMQ_EXCHANGE_NAME}
rabbitmq.routing.key=${RABBITMQ_ROUTING_KEY}
rabbitmq.queue.name.log=${RABBITMQ_LOG_QUEUE}
```

Kode 3.9: Konfigurasi Brevo dan RabbitMQ di file *application.properties*

A Pengambilan Konfigurasi dan Inisialisasi *BrevoService*

Kelas *BrevoService* berfungsi sebagai komponen utama untuk mengelola komunikasi dengan vendor layanan email Brevo. Untuk menginisialisasi kelas ini, konfigurasi seperti *API key* dan URL disimpan di dalam file *application.properties* dan diambil menggunakan teknik *dependency injection* milik Spring Boot. Konfigurasi ini memungkinkan pemisahan antara logika program dengan informasi sensitif atau yang dapat berubah tergantung lingkungan, seperti *development* atau *production*. Parameter yang diinject ke dalam konstruktor antara lain adalah *restTemplate* yang digunakan untuk melakukan permintaan HTTP, *brevoApiKey* yang berisi kunci autentikasi untuk mengakses layanan Brevo, serta *brevoUrl* yang merupakan endpoint tujuan. Nilai dari *brevoApiKey* dan *brevoUrl* dipetakan dari variabel di *application.properties*, yaitu *vendor.email.brevo.api.key* dan *vendor.email.brevo.api.url*. Kode *Java* berikut menunjukkan struktur inisialisasi tersebut seperti terlihat pada Kode 3.10.

```
@Service
public class BrevoService {
    public BrevoService(RestTemplate restTemplate,
                        @Value("${vendor.email.brevo.api.key}")
   String brevoApiKey,
```

```
5                         @Value("${vendor.email.brevo.api.url}")
    String brevoUrl) {
6         this.restTemplate = restTemplate;
7         this.brevoApiKey = brevoApiKey;
8         this.brevoUrl = brevoUrl;
9     }
10 }
```

Kode 3.10: Inisialisasi *BrevoService* dengan parameter konfigurasi

B Struktur *Payload* dan Jenis Permintaan ke Brevo

Layanan pengiriman email melalui Brevo mendukung berbagai jenis *payload* berdasarkan kebutuhan pengguna. Pengiriman dapat dilakukan untuk satu email, email dengan konten HTML, pengiriman massal, serta email dengan lampiran. Format yang digunakan bervariasi antara *application/json* untuk *payload* biasa dan *multipart/form-data* untuk email dengan lampiran.

- Email teks biasa (non-HTML)
  Payload sederhana tanpa format HTML terlihat pada Kode3.11. Nantinya ini digunakan untuk pengiriman internal seperti notifikasi sistem.

```
1  {
2   "recipientEmail": "user1@gmail.com",
3   "sender": "",
4   "useCase": "Notifikasi terakhir pembayaran",
5   "content": "Halo ini tes email"
6  }
7
```

Kode 3.11: Contoh payload email teks biasa

- Email dengan konten HTML
  Perbedaanya dengan yang biasa, terlihat pada Kode 3.12 itu diiisi konten diformat dengan tag HTML, digunakan untuk email promosi atau undangan.

```
1 {
2   "recipientEmail": "user1@gmail.com",
3   "sender": "",
4   "useCase": "UNDANGAN",
5   "content": "<!DOCTYPE html><html><body><h1>Undangan</h1><p>
      Anda diundang ke acara kami.</p></body></html>"
6 }
```

```

```

Kode 3.12: Contoh payload email HTML

- Pengiriman bulk (isi berbeda)
  Mengirim banyak email dengan konten berbeda untuk masing-masing penerima. Format berupa array objek JSON seperti terlihat pada Kode 3.13. Penggunaannya untuk email konfirmasi pendaftaran ke banyak peserta acara VIP karena isinya biasanya harus dibedakan.

```
[
  {
    "recipientEmail": "tamuvip1@example.com",
    "sender": "",
    "useCase": "INFO",
    "content": "Informasi penting A"
  },
  {
    "recipientEmail": "tamuvip2@example.com",
    "sender": "",
    "useCase": "INFO",
    "content": "<html><body>Informasi penting B</body></html
    >"
  }
]

```

Kode 3.13: Contoh payload bulk dengan konten berbeda

- Satu email dengan lampiran
  Menggunakan format *multipart/form-data*, terdiri dari JSON metadata isi JSON terlihat pada Kode 3.14 dan satu atau lebih file. Digunakan saat perlu melampirkan dokumen. Contoh file terlampirnya itu invoice.pdf untuk pelanggan Minyma.

```
{
  "recipientEmail": "user1@gmail.com",
  "sender": "",
  "useCase": "Invoice Paket Pernikahan",
  "content": "Ini isi emailnya"
}

```

Kode 3.14: Contoh JSON untuk satu email dengan lampiran

- Bulk email dengan lampiran yang sama
  Mengirim email ke banyak penerima, konten sama, lampiran sama. Isi JSON terlihat pada Kode 3.15 dan bisa satu atau lebih file dengan format menggunakan multipart JSON untuk data dan file. Penggunaannya untuk mengirim email undangan ke para tamu yang diundang oleh pelanggan Minyma.

```
{
  "sender": "jayasantosoteknologi@gmail.com",
  "useCase": "Undangan Pernikahan",
  "content": "<h1>Pernikahan A&L </h1><p>Tempat dan Waktu di
   lampirkan!</p>",
  "recipientEmails": [
    "tamu1@gmail.com",
    "tamu2@gmail.com"
  ]
}

```

Kode 3.15: Contoh payload JSON bulk dengan lampiran

Setiap jenis permintaan ini ditangani oleh *endpoint* yang berbeda pada sistem *backend*, tergantung pada kebutuhan. File lampiran akan dikonversi ke format *Base64* dan dikirim bersama *payload* ke API Brevo. Jika terjadi kesalahan, sistem akan melempar *BrevoApiException* dan memberikan respon *error* ke pengguna.

C Alur Umum Proses Pengiriman Email ke Brevo

Pengiriman email ke Brevo mengikuti pola umum yang seragam, baik untuk satu email maupun pengiriman massal, dengan atau tanpa lampiran. Inti dari proses ini adalah membentuk *payload* sesuai dengan format yang diterima oleh API Brevo, melakukan validasi, menangani file jika ada, dan memanggil API melalui *RestTemplate*. Apabila terjadi kesalahan selama pengiriman, sistem akan menangkapnya dan melempar *BrevoApiException*. Pseudocode proses pengiriman email ditunjukkan pada Kode 3.16.

```
PROCEDURE SendEmailToBrevo(request)

    IF request IS INVALID THEN
        RETURN HTTP 400 BAD REQUEST
```

```
    ENDIF

    IF request CONTAINS FILES THEN
        FOR EACH file IN request.files DO
            ENCODE file TO Base64
            ADD TO payload.attachments
        ENDFOR
    ENDIF

    IF request.content CONTAINS "<html>" THEN
        payload.htmlContent = request.content
    ELSE
        payload.textContent = request.content
    ENDIF

    SET payload.sender = request.sender
    SET payload.to = request.recipients

    TRY
        CALL BrevoAPI(payload)
        SAVE email TO database
        RETURN HTTP 200 OK
    CATCH BrevoApiException
        RETURN HTTP 500 INTERNAL SERVER ERROR
    ENDTRY

END PROCEDURE
```

Kode 3.16: Pseudocode umum proses pengiriman email ke Brevo

D Pengiriman Email melalui RabbitMQ

Pengiriman email dilakukan secara asinkron menggunakan RabbitMQ sebagai sistem antrean. Hal ini memungkinkan proses pengiriman dilakukan secara terpisah dari permintaan utama, sehingga meningkatkan efisiensi dan keandalan sistem. Alur umum dari proses ini adalah sebagai berikut:

1. Email dan lampiran disimpan ke basis data lokal terlebih dahulu dengan status *PENDING* seperti terlihat pada Kode 3.17. Penyimpanan dilakukan melalui *JdbcEmailRepository* dan *JdbcAttachmentRepository*.

```
FUNCTION savePendingEmailWithAttachments(message, attachments
    ):
```

```
    IF sender IS NULL:
        SET sender TO defaultSender
    SET message.status TO "PENDING"
    SET message.createdAt TO now()
    SET idEmail TO SAVE message TO database
    FOR EACH attachment IN attachments:
        SET attachment.emailId TO idEmail
    SAVE attachments TO database
END FUNCTION

```

Kode 3.17: Pseudocode penyimpanan email dan lampiran ke database

2. Sistem akan membentuk objek *EmailWithAttachments* yang berisi email dan daftar lampiran. Objek ini akan dikirim ke antrean RabbitMQ menggunakan metode *convertAndSend()*.

3. Sebelum dikirim, setiap lampiran dikonversi dari data biner menjadi string teks menggunakan metode *Base64*. Konversi ini diperlukan agar file dapat dikirim dalam format *JSON* melalui protokol yang berbasis teks seperti HTTP dan message queue. Contoh implementasi konversi data lampiran ke *Base64* ditunjukkan pada Kode 3.18.

```
    import java.util.Base64;

    byte[] fileBytes = attachment.getFileData();
    String base64Encoded = Base64.getEncoder().encodeToString
    (fileBytes);

```

Kode 3.18: Contoh konversi data biner lampiran ke Base64 di Java

4. Setelah semua lampiran dikonversi, sistem akan mengirim objek *EmailWithAttachments* ke *exchange* dengan *routing key* tertentu yang telah ditentukan di konfigurasi aplikasi seperti terlihat pada Kode 3.19.

```
FUNCTION sendEmailWithAttachmentsToQueue(emailWithAttachments
    ):
    FOR EACH attachment IN emailWithAttachments.attachments:
        base64Content = encodeToBase64(attachment.fileData)
        attachment.fileData = base64Content
    LOG "Sending email to queue"
    CALL rabbitTemplate.convertAndSend(exchange, routingKey,
    emailWithAttachments)
```

```
END FUNCTION

```

Kode 3.19: Pseudocode pengiriman email ke antrean RabbitMQ

5. Konsumen dari antrean akan mendengarkan isi *queue*, lalu mengambil pesan yang masuk. Setelah berhasil diambil, sistem akan mencoba mengirim email ke vendor Brevo.

6. Jika pengiriman ke Brevo berhasil, maka status email akan diperbarui dari *PENDING* menjadi *SENT*. Namun jika gagal (misalnya karena jaringan atau respons dari vendor), status akan menjadi *FAILED* dan sistem akan mencatat pesan kesalahan, logika terlihat pada Kode 3.20.

```
FUNCTION consumeEmailFromQueue(message):
    PARSE message TO EmailWithAttachments
    TRY:
        CALL brevoService.sendEmailWithAttachments(message)
        UPDATE status TO "SENT" IN database
    CATCH Exception e:
        LOG error message
        UPDATE status TO "FAILED" IN database
END FUNCTION

```

Kode 3.20: Pseudocode proses konsumsi dan pengiriman dari antrean

Dengan pendekatan ini, proses pengiriman email dapat tetap berjalan meskipun terjadi lonjakan permintaan, karena seluruh beban dikelola melalui antrean dan dieksekusi secara terpisah oleh konsumen RabbitMQ.

E Struktur *Package* pada Modul Brevo

Struktur kode pada modul *backend* untuk integrasi dengan Brevo dirancang secara modular agar memisahkan tanggung jawab tiap komponen. Pembagian ini memudahkan pemeliharaan, pengujian, dan pengembangan lebih lanjut. Berikut penjabaran fungsi dari masing-masing *package* yang digunakan:

- *config*: Menyimpan konfigurasi sistem seperti *AppConfig*, *OpenApiConfig*, dan *RabbitMQConfig*.

- *controller*: Menangani permintaan API dari klien. Pada modul ini, controller bertugas menerima permintaan pengiriman email, baik satuan maupun massal, dengan atau tanpa lampiran, kemudian mengarahkan pesan tersebut ke antrean.
- *dto* (Data Transfer Object): Menyediakan struktur data perantara antara lapisan controller dan service, seperti *EmailRequestDTO*, *BulkEmailRequestDTO*, dan *EmailResponseDTO*.
- *exception*: Menangani pengecualian yang dapat terjadi selama proses pengiriman email, seperti *BrevoApiException* ketika terjadi kegagalan integrasi ke API Brevo, serta *AttachmentProcessingException* untuk kegagalan pemrosesan file lampiran.
- *model*: Mewakili entitas utama yang disimpan ke dalam basis data, yaitu objek *Email* dan *Attachment*.
- *publisher*: Bertanggung jawab untuk mengirim pesan ke antrean RabbitMQ, menggunakan komponen *RabbitMQProducer*.
- *consumer*: Bertugas mengambil pesan dari antrean, memproses isinya, dan mengirimkannya ke API Brevo. Paket ini menjadi bagian utama dari logika pengiriman asinkron.
- *repository*: Mengatur proses penyimpanan dan pengambilan data dari basis data menggunakan pendekatan JDBC. Dua repositori utama adalah *JdbcEmailRepository* dan *JdbcAttachmentRepository*.
- *service*: Mewadahi logika bisnis utama, seperti validasi isi email, konversi lampiran ke format *Base64*, pemanggilan API, dan penyimpanan data. Kelas *EmailService* menjadi pusat pengelolaan proses ini.

F Penanganan Kesalahan, *Logging*, dan Peran *Debugging*

Untuk menjaga stabilitas sistem, seluruh kesalahan yang terjadi pada proses pengiriman email atau pengolahan lampiran akan ditangani secara global menggunakan *GlobalExceptionHandler*. Seluruh pesan kesalahan akan dicatat menggunakan mekanisme *logging*, sementara proses *debugging* dilakukan berdasarkan log tersebut. Kode 3.21 menunjukkan pseudocode penanganan *error* secara global dan pencatatan log.

```
@ExceptionHandler(CustomException)
function handleCustomException(ex):
    log warning "Custom exception occurred: " + ex.message
    return BAD_REQUEST with message and code

@ExceptionHandler(Exception)
function handleUnexpected(ex):
    log error "Unexpected error: " + ex.message
    return INTERNAL_SERVER_ERROR with detail
```

Kode 3.21: Pseudocode penanganan kesalahan dan pencatatan log global

Jenis *exception* yang ditangani dalam sistem adalah sebagai berikut:

- *CustomException*: Superclass dari seluruh *exception* kustom. Digunakan untuk mengembalikan pesan dan kode kesalahan spesifik.
- *AttachmentProcessingException*: Menangani kesalahan saat pemrosesan lampiran, seperti file rusak atau tidak bisa dikonversi ke *Base64*.
- *BrevoApiException*: Menangani kegagalan komunikasi dengan vendor Brevo, seperti *invalid API key* atau masalah koneksi.
- *NotFoundException*: Dilemparkan saat email tidak ditemukan berdasarkan *id*.
- *Exception*: Menangani semua *error* tak terduga sebagai fallback.

Pengujian proses pengiriman email dilakukan menggunakan metode *debugging* dengan menangkap kesalahan yang mungkin terjadi, kemudian mencetak informasi *error* dan jejak kesalahan ke log. Implementasi pseudocode pengujian ini ditunjukkan pada Kode 3.22, di mana sistem akan mencetak pesan keberhasilan atau kesalahan berdasarkan hasil eksekusi fungsi *sendEmail*.

```
function testEmailSending():
    try:
        result = emailService.sendEmail(testPayload)
        assert result.status == "SENT"
        log info "Email test passed"
    catch (Exception e):
        log error "Email test failed: " + e.message
        debugPrint(e.stackTrace)
```

Kode 3.22: Contoh pseudocode debugging sederhana saat pengujian email

Dengan struktur ini, setiap *error* yang terjadi akan tercatat dan bisa dianalisis lebih lanjut menggunakan informasi log.

G Pengujian *Endpoint* Brevo

Setelah seluruh *endpoint* Brevo diimplementasikan, dilakukan pengujian menggunakan Postman untuk memastikan setiap fungsionalitas berjalan sesuai spesifikasi. Pengujian mencakup skenario sukses maupun gagal, dengan mengevaluasi status respons, isi pesan balik, serta validitas data yang diproses. Tabel 3.9 menyajikan daftar seluruh *endpoint* yang telah diuji, beserta metode HTTP dan deskripsi fungsi masing-masing. Selanjutnya, hasil pengujian secara rinci untuk setiap *endpoint* akan disajikan melalui dokumentasi gambar pada bagian berikutnya.

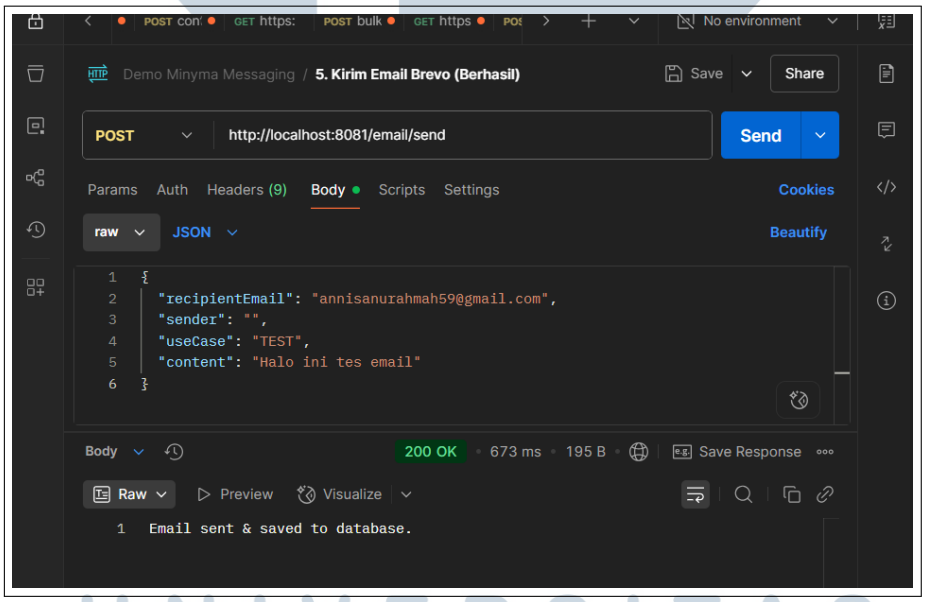
Tabel 3.9. Daftar *endpoint* Brevo dan deskripsinya

No	Nama Endpoint	Method dan URL	Deskripsi
1	Kirim Email (Brevo)	POST /email/send	Mengirim email tanpa lampiran melalui vendor Brevo.
2	Kirim Email Bulk (Berbeda)	POST /email/send/bulk	Mengirim banyak email berbeda satu per satu tanpa lampiran.
3	Kirim Email Bulk + Lampiran	POST /email/send/bulk-emailwithattachment	Mengirim email ke banyak penerima dengan konten dan lampiran yang sama.
4	Kirim Email + Lampiran	POST /email/send-with-attachments	Mengirim satu email dengan satu atau lebih file lampiran.
5	Ambil Email by ID (Brevo)	GET /email/{idEmail}	Mendapatkan detail email dari database berdasarkan ID.
6	Update Email (Brevo)	PUT /email/{idEmail}	Memperbarui konten email Brevo berdasarkan ID.
Lanjutan pada halaman berikutnya			

Daftar *endpoint* Brevo dan deskripsinya (lanjutan)

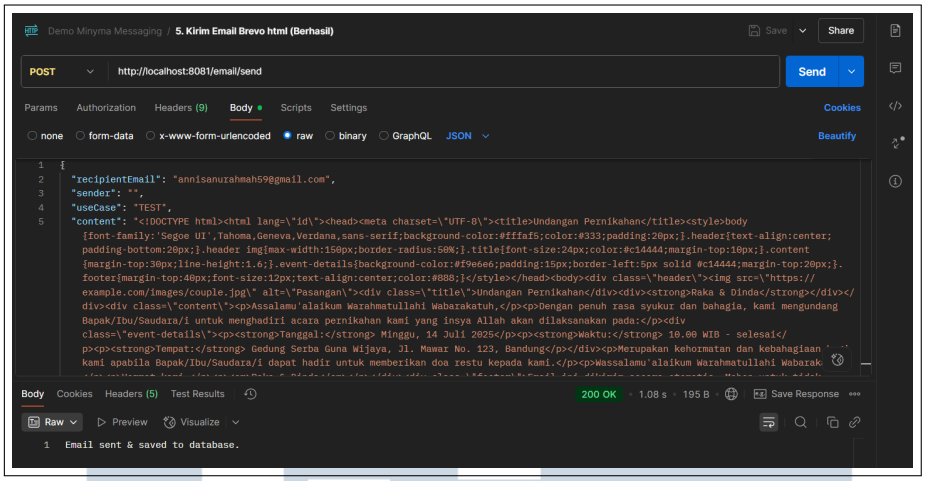
No	Nama Endpoint	Method dan URL	Deskripsi
7	Hapus Email (Brevo)	DELETE /email/{idEmail}	Menghapus email dari database berdasarkan ID.
8	Ambil Semua Email	GET /email	Mengambil seluruh data email yang telah dikirim.

Endpoint POST /email/send digunakan untuk mengirim email menggunakan layanan Brevo. Pada pengujian pertama, konten email berupa teks biasa dikirim melalui Postman. Sistem memberikan respons sukses dengan status *200 OK*, menandakan bahwa pengiriman berjalan tanpa *error*. Hasil pengujian ini ditunjukkan pada Gambar 3.13.



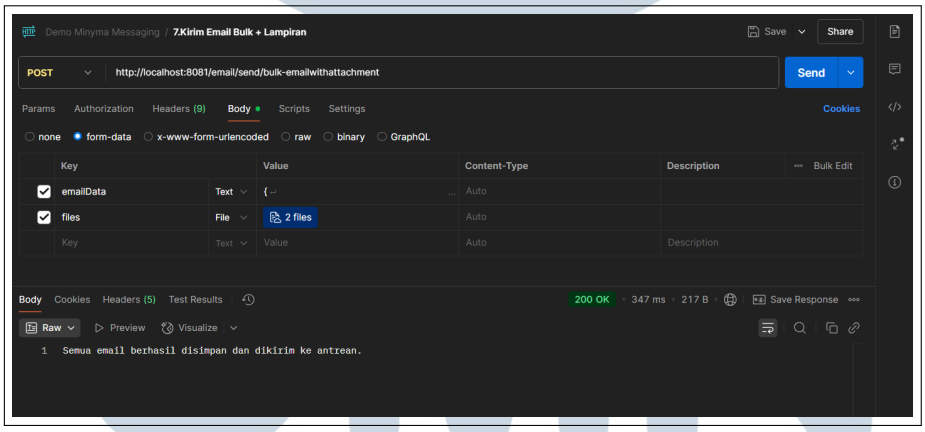
Gambar 3.13. Pengujian *endpoint* 1 - kirim email teks biasa menggunakan Brevo

Selanjutnya, pengujian dilakukan dengan mengubah konten email menjadi format HTML, seperti undangan digital. Gambar 3.23 menunjukkan bahwa sistem tetap berhasil memproses permintaan dan menampilkan respons sukses di Postman.



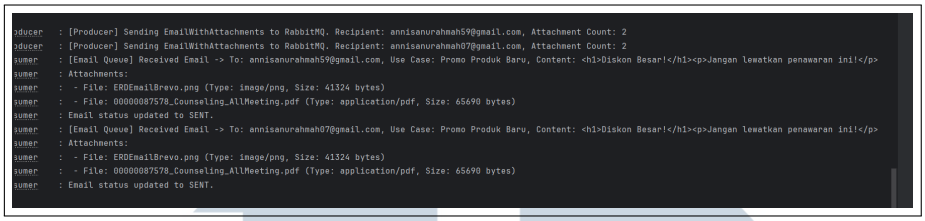
Gambar 3.14. Pengujian *endpoint* 1 - kirim email HTML menggunakan Brevo

Endpoint POST /email/send/bulk-emailwithattachment digunakan untuk mengirim email beserta lampiran ke banyak penerima sekaligus. Pengujian menunjukkan bahwa proses pengiriman berjalan dengan sukses tanpa *error*. Gambar 3.15 menunjukkan hasil pengujian endpoint ini melalui Postman.



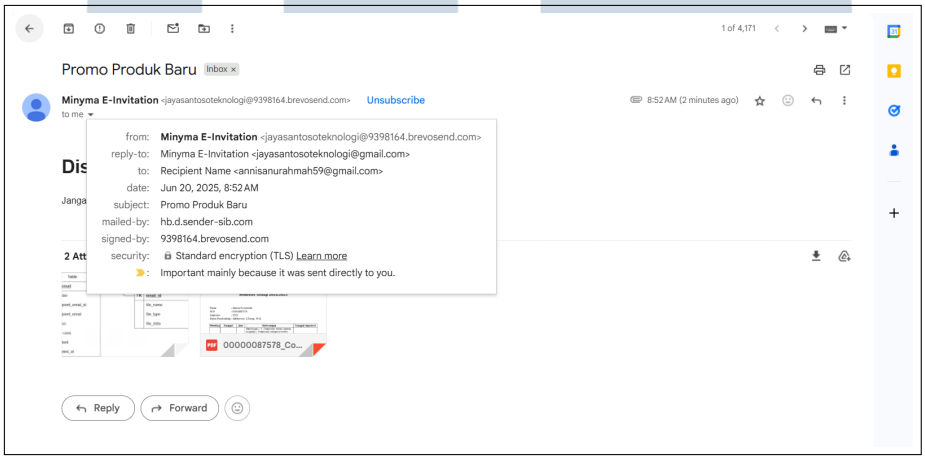
Gambar 3.15. Pengujian *endpoint* 3 mengirim email massal dengan lampiran

Log konsol pada Gambar 3.16 memperlihatkan bahwa proses pengiriman dicatat dan masuk ke antrean RabbitMQ sebelum diteruskan ke layanan pengiriman email.

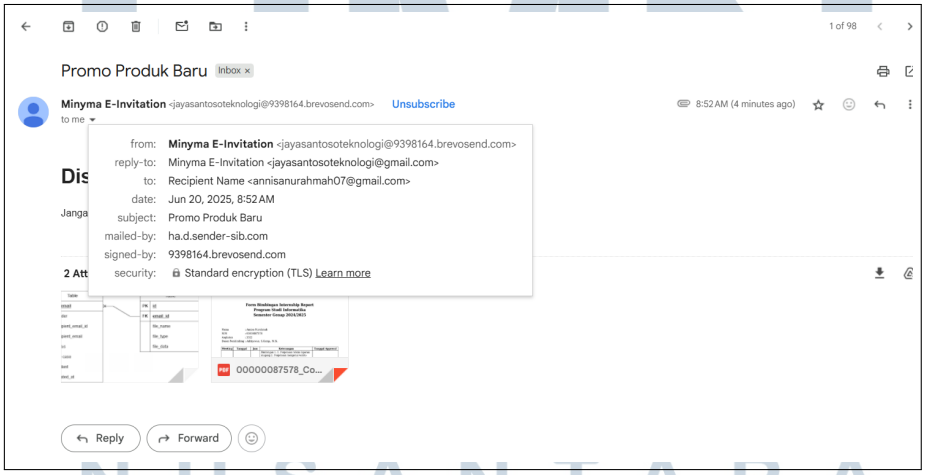


Gambar 3.16. Log antrean RabbitMQ saat pengiriman email massal dengan lampiran

Gambar 3.17 dan Gambar 3.18 memperlihatkan bahwa dua email diterima di dua akun Gmail berbeda, masing-masing dengan lampiran yang sesuai.



Gambar 3.17. Email pertama diterima dengan file lampiran



Gambar 3.18. Email kedua diterima oleh penerima berbeda dengan lampiran utuh

Pengujian endpoint lainnya seperti *GET /email/{idEmail}*, *PUT /email/{idEmail}*, dan *DELETE /email/{idEmail}* menunjukkan hasil yang

sesuai dengan ekspektasi, dengan data dapat ditampilkan, diperbarui, dan dihapus dari basis data secara normal. *Endpoint GET /email* juga berhasil menampilkan daftar seluruh email yang telah dikirim.

3.3.4 Pengembangan Fitur *Contact Us*

Fitur *Contact Us* dikembangkan untuk memberikan jalur komunikasi langsung antara pengguna dan admin melalui email. Proses dimulai ketika pengguna mengisi formulir pada antarmuka *frontend*, lalu data dikirim ke *backend* melalui permintaan *POST* ke *endpoint /email/contact/send*. Permintaan ini diterima oleh bagian *controller*, yang kemudian meneruskannya ke *service* untuk diproses lebih lanjut. Di dalam *service*, terdapat dua proses utama, yaitu pengiriman email ke admin dan penyimpanan informasi pesan ke dalam basis data menggunakan JDBC.

A Konfigurasi SMTP Gmail

Pengiriman email dilakukan melalui protokol SMTP Gmail. Konfigurasi SMTP tersebut disimpan dalam berkas *application.properties* yang berada di konfigurasi global sistem, karena juga digunakan oleh fitur pengiriman lainnya. Konfigurasi ini ditunjukkan pada Kode 3.23.

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=${GMAIL_USERNAME}
spring.mail.password=${GMAIL_PASSWORD}
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Kode 3.23: Konfigurasi *application.properties* untuk *Contact Us*

B Payload Permintaan

Payload permintaan dari pengguna dikirim dalam format *JSON* yang berisi informasi yang dibutuhkan untuk menghubungi layanan melalui fitur *Contact Us*. Struktur data ini memuat empat atribut utama: *firstName*, *lastName*, *email*, dan *message*. Masing-masing atribut mewakili informasi identitas pengguna dan isi pesan yang ingin disampaikan kepada pengelola sistem.

Sebagai contoh, pada Kode 3.24 ditunjukkan bentuk *payload* yang dikirimkan ketika seorang pengguna mengisi formulir *Contact Us* untuk bertanya

lebih lanjut mengenai fitur aplikasi Minyma. Atribut *firstName* dan *lastName* mencerminkan nama lengkap pengguna, *email* digunakan sebagai alamat pengirim untuk keperluan balasan, dan *message* berisi konten pertanyaan atau komentar dari pengguna.

```
{
"firstName": "Annisa",
"lastName": "Nurahmah",
"email": "annisanurahmah@gmail.com",
"message": "Halo, saya ingin mengetahui lebih lanjut tentang fitur
    Minyma."
}
```

Kode 3.24: Contoh *payload* permintaan *Contact Us*

Payload ini akan diterima oleh *controller backend*, kemudian diteruskan untuk dicatat di basis data.

C Pengiriman Email dan Penyimpanan Pesan

Untuk melakukan pengiriman email, *backend* menggunakan *JavaMailSender*. Email dikirim dari akun sistem ke alamat email admin, sementara bagian *reply-to* diatur ke alamat email milik pengguna. Pseudocode proses pengiriman email ditunjukkan pada Kode 3.25.

```
function processContactMessage(request):
    create email message using mailSender
    set to = senderEmail
    set from = senderEmail
    set reply-to = request.email
    set subject = "Mynima Contact Us from " + request.firstName
    set body = request.message
    send email
    insert data to database
    return OK response
```

Kode 3.25: Pseudocode pengiriman email *Contact Us*

Setelah email berhasil dikirim, sistem melanjutkan proses dengan mencatat pesan ke dalam basis data. Pencatatan data dilakukan menggunakan pendekatan JDBC. Karena tidak menggunakan ORM seperti JPA, pernyataan SQL disusun secara manual dan dieksekusi menggunakan objek *JdbcTemplate*. Penggunaan JDBC seperti pada Kode 3.26 memberikan kontrol penuh atas eksekusi perintah

SQL, serta efisien untuk kasus penyimpanan sederhana. Namun, pendekatan ini juga menuntut penanganan *error* eksplisit agar sistem dapat memberikan respons yang tepat apabila terjadi kegagalan saat eksekusi.

```
String sql = "INSERT INTO contact_us (first_name, last_name, email
    , message, created_at) " +
"VALUES (?, ?, ?, ?, ?)";

jdbcTemplate.update(sql,
request.getFirstName(),
request.getLastName(),
request.getEmail(),
request.getMessage(),
LocalDateTime.now()
);
```

Kode 3.26: Penyimpanan data *Contact Us* melalui JdbcTemplate

D Struktur *Package* Fitur *Contact Us*

Struktur kode *backend* untuk fitur *Contact Us* diorganisasi ke dalam beberapa bagian utama dengan tanggung jawab sebagai berikut:

- *Controller*: Menangani permintaan yang datang dari sisi *frontend* dan meneruskannya ke lapisan *service*.
- *DTO*: Menyimpan struktur data yang merepresentasikan *payload* permintaan dari pengguna.
- *Model*: Merepresentasikan entitas pesan atau kontak yang akan disimpan dalam basis data.
- *Repository*: Mengatur interaksi dengan basis data menggunakan *JdbcTemplate*, termasuk penyimpanan dan pengambilan data.
- *Service*: Menangani logika bisnis utama, seperti mengirim email notifikasi dan mencatat pesan ke dalam basis data.

Struktur ini membantu memisahkan tanggung jawab secara jelas, mendukung keterbacaan kode, dan memudahkan proses pemeliharaan serta pengujian selama pengembangan sistem.

E Penanganan Kesalahan pada Fitur *Contact Us*

Untuk menjaga stabilitas sistem, pengiriman email dan pencatatan pesan dibungkus dalam blok *try-catch*. Apabila terjadi kesalahan saat mengirim email (*MessagingException*) atau saat eksekusi SQL (*Exception*), sistem akan mengembalikan respons dengan status *500 Internal Server Error* dan pesan sesuai penyebab.

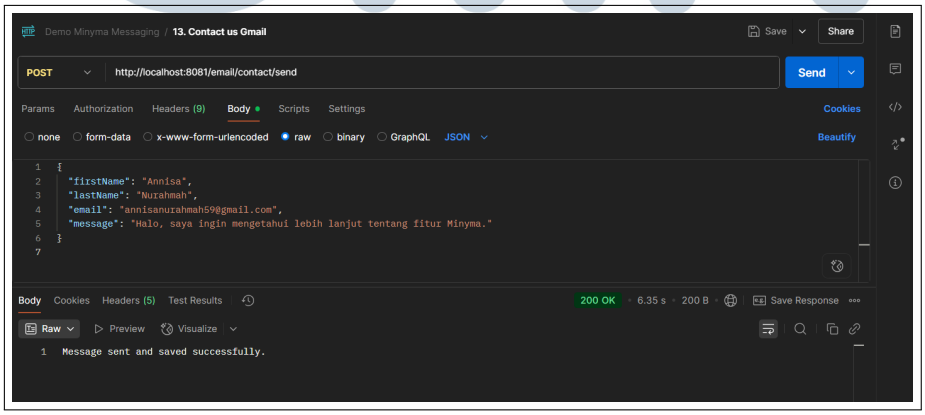
```
try:
    send email via mailSender
    save message to database
    return OK response
catch MessagingException:
    return error response: Messaging error
catch Exception:
    return error response: General error
```

Kode 3.27: Penanganan error saat pengiriman Contact Us

F Pengujian Endpoint *Contact Us*

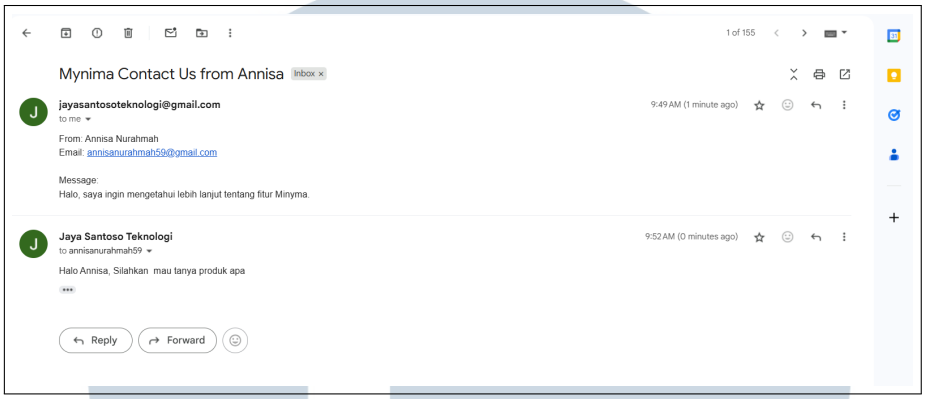
Pengujian dilakukan menggunakan Postman dengan skenario pengiriman pesan sukses. Pada pengujian sukses, sistem merespons dengan status *200 OK* dan pesan sukses. Gambar pengujian ditampilkan pada bagian Gaambar 3.19.

Endpoint POST /email/contact/send berhasil digunakan untuk mengirim pesan dari form kontak menggunakan layanan Gmail SMTP ke alamat email admin. Pengujian awal terhadap *endpoint* ini dilakukan menggunakan Postman seperti terlihat pada Gambar 3.19, yang memperlihatkan bahwa request berhasil diproses oleh sistem.



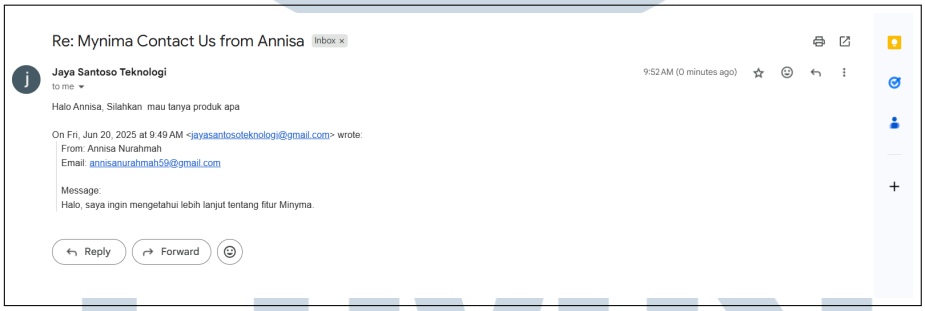
Gambar 3.19. Pengujian *endpoint Contact Us* melalui Postman

Selanjutnya, Gambar 3.20 menunjukkan bahwa email dari form kontak masuk ke inbox admin dengan format yang sesuai.



Gambar 3.20. Email masuk ke admin dari *form contact*

Gambar 3.21 memperlihatkan balasan email dari admin ke pengirim form. Proses ini mengonfirmasi bahwa pengiriman email melalui SMTP berjalan dua arah, dan sistem berhasil meneruskan pesan balasan kepada pengguna.



Gambar 3.21. Balasan email admin ke pengirim *form contact*

3.3.5 Implementasi Dokumentasi Swagger API

Untuk memudahkan proses integrasi dan pemahaman teknis oleh tim pengembang lain, sistem *Messaging Service* dilengkapi dengan dokumentasi otomatis menggunakan Swagger UI. Swagger menyediakan tampilan antarmuka visual interaktif yang mendokumentasikan seluruh *endpoint* API, termasuk metode HTTP, format *request body*, parameter, dan struktur response. Dokumentasi ini sangat membantu tim *frontend* dalam memahami struktur data dan alur integrasi dengan *backend*.

A Konfigurasi Swagger OpenAPI

Integrasi Swagger dilakukan dengan menambahkan dependensi *springdoc-openapi-starter-webmvc-ui* ke dalam berkas *pom.xml* seperti pada Kode 3.28

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.8.8</version>
</dependency>
```

Kode 3.28: Dependensi Swagger di pom.xml

Selanjutnya, konfigurasi dokumentasi API dilakukan menggunakan Swagger melalui pustaka *OpenAPI*. Pada proyek ini, Swagger dikonfigurasi dalam kelas konfigurasi bernama *OpenApiConfig* sebagaimana ditunjukkan pada Kode 3.29. Kelas ini berfungsi untuk menetapkan metadata dasar mengenai layanan yang dikembangkan. Di dalam kelas *OpenApiConfig*, didefinisikan sebuah *bean* bernama *messagingOpenAPI()* yang menghasilkan objek *OpenAPI*. Dalam konteks *Spring Framework*, *bean* adalah objek yang dikelola oleh *Spring container* dan dapat digunakan secara global di dalam aplikasi. Dengan mendefinisikan *bean*, objek tersebut dapat disuntikkan ke komponen lain yang membutuhkan, tanpa harus membuat ulang instansinya secara manual. Objek *OpenAPI* tersebut memuat informasi seperti judul dokumentasi (*title*), deskripsi layanan (*description*), dan versi API (*version*). Dalam konfigurasi ini, judul yang digunakan adalah "*Mynima Messaging API*", dengan deskripsi "*API Documentation for Minyma Messaging Services*", dan versi "*v1.0*". Konfigurasi ini memungkinkan Swagger untuk menghasilkan dokumentasi API secara otomatis, yang dapat diakses melalui antarmuka berbasis web. Dokumentasi ini sangat membantu dalam proses pengujian dan integrasi sistem oleh tim pengembang karena menyediakan tampilan interaktif dari seluruh *endpoint* yang tersedia.

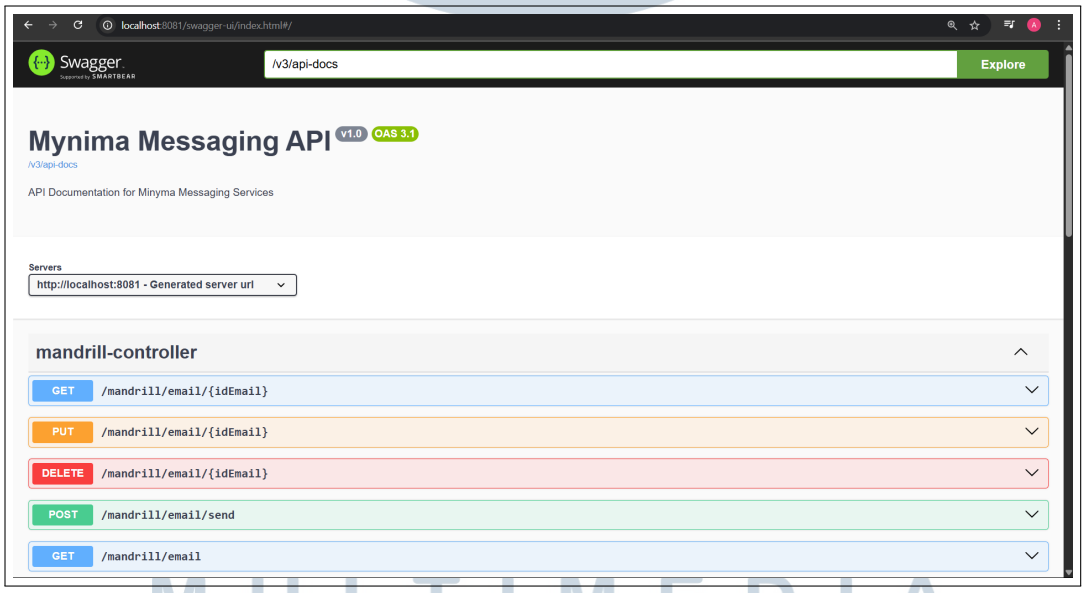
```
package com.mynima.messaging.config;

import io.swagger.v3.oas.models.OpenAPI;
import io.swagger.v3.oas.models.info.Info;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OpenApiConfig {

```

```
11      @Bean
12      public OpenAPI messagingOpenAPI() {
13          return new OpenAPI()
14                  .info(new Info()
15                          .title("Mynima Messaging API")
16                          .description("API Documentation for Minyma
        Messaging Services")
17                          .version("v1.0"));
18      }
19  }
```

Kode 3.29: Konfigurasi Swagger OpenAPI

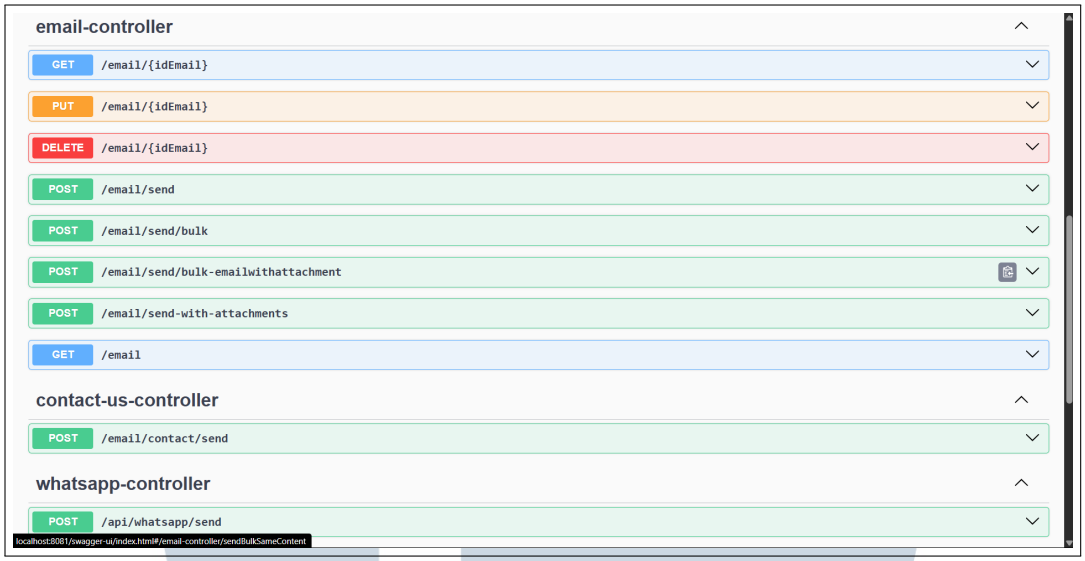
B Tampilan Swagger untuk Berbagai Endpoint

Pada Gambar 3.22 ditampilkan antarmuka Swagger untuk *endpoint* pengiriman email melalui Mandrill. Walaupun Mandrill tidak lagi digunakan sebagai vendor utama, dokumentasinya tetap tersedia untuk kebutuhan historis dan pengujian internal.



Gambar 3.22. Tampilan Swagger untuk *endpoint* Mandrill

Gambar 3.23 menampilkan *endpoint* utama untuk layanan Brevo, pengiriman pesan masal, pengiriman dengan lampiran, fitur *Contact Us* yang menggunakan SMTP Gmail, dan layanan WhatsApp melalui Meta API.



Gambar 3.23. Tampilan Swagger untuk *endpoint* Brevo, *Contact Us*, dan WhatsApp

Selama proses integrasi, dilakukan diskusi dengan tim *frontend* untuk menyelaraskan struktur dokumentasi dengan kebutuhan antarmuka pengguna. Dengan Swagger, tim frontend dapat langsung mengeksplorasi setiap *endpoint* dan memahami struktur data tanpa harus membaca kode *backend* secara langsung. Meskipun Swagger menyediakan antarmuka uji coba, proses pengujian utama tetap dilakukan menggunakan Postman. Hal ini karena Postman mendukung fitur lanjutan seperti pengelolaan token dinamis, pengujian berulang, dan pengaturan uji kompleks seperti file dan pengiriman email masal.

3.3.6 Implementasi Layanan WhatsApp

Layanan WhatsApp dikembangkan untuk mendukung pengiriman pesan melalui platform WhatsApp Business API dari Meta. Proses integrasi dimulai dari pembuatan akun Meta Developer, setup aplikasi dan nomor bisnis, hingga pengujian pengiriman menggunakan token sementara.

A Konfigurasi WhatsApp API

Konfigurasi awal API WhatsApp dilakukan melalui file *application.properties* dengan parameter sebagai berikut, Kode 3.30.

```
#whatsapp
whatsapp.api.url=${WHATSAPP_API_URL}
```

```
whatsapp.api.token=${WHATSAPP_API_TOKEN}
```

Kode 3.30: Konfigurasi WhatsApp di *application.properties*

Nilai konfigurasi seperti URL dan token disimpan sebagai variabel lingkungan untuk alasan keamanan dan fleksibilitas deployment.

B Payload Pengiriman WhatsApp

Format permintaan untuk mengirim pesan WhatsApp mencakup berbagai jenis pesan, seperti teks dan dokumen. Contoh payload untuk pengiriman terlihat pada Kode 3.31

```

{
  "recipientNumber": "6281234567890",
  "messageType": "text",
  "messageBody": "Halo, ini adalah pesan dari Mynima",
  "previewUrl": false
}
```

Kode 3.31: Contoh *payload* pesan WhatsApp

Untuk pengiriman dokumen, field tambahan seperti *mediaId*, *caption*, dan *mediaFilename* juga digunakan.

C Pengiriman Pesan WhatsApp dengan Token API

Proses pengiriman pesan WhatsApp dimulai dengan pengambilan konfigurasi dari *application.properties*, yaitu *apiUrl* dan *token*. Setelah itu, sistem membentuk struktur permintaan HTTP dan mengirimkannya ke endpoint WhatsApp API. Alur pengiriman ditunjukkan melalui pseudocode Kode 3.32

```
Ambil nilai API_URL dan TOKEN dari konfigurasi

Buat objek HEADER
  HEADER.ContentType = application/json
  HEADER.Authorization = Bearer TOKEN

Buat BODY permintaan sesuai jenis pesan:
  Jika messageType = "text":
    BODY = {
      "messaging_product": "whatsapp",
      "to": recipientNumber,
```

```
      "type": "text",
      "text": {
        "body": messageBody,
        "preview_url": previewUrl
      }
    }

Kirim POST request ke API_URL dengan HEADER dan BODY
Simpan response dari server
```

Kode 3.32: Pseudocode pengiriman pesan WhatsApp

D Pencatatan dan Penanganan Respons Pengiriman

Setelah permintaan dikirim, sistem menangani respons yang dikembalikan oleh server. Respons ini kemudian dicatat ke dalam basis data, baik berhasil maupun gagal. Proses *logging* ini penting untuk *debugging*, pseudocode pencatatan terlihat pada Kode 3.33

```

Inisialisasi objek Message
  Message.recipientNumber = recipientNumber
  Message.messageType = messageType
  Message.messageBody = messageBody
  Message.previewUrl = previewUrl
  Message.createdAt = now()
  Message.updatedAt = now()

IF response.statusCode is 2xx THEN
  Message.status = "SENT"
  Message.apiResponse = response.body
ELSE
  Message.status = "FAILED"
  Message.apiResponse = response.errorMessage
END IF

Simpan Message ke database
```

Kode 3.33: Pseudocode *logging* dan penanganan respons

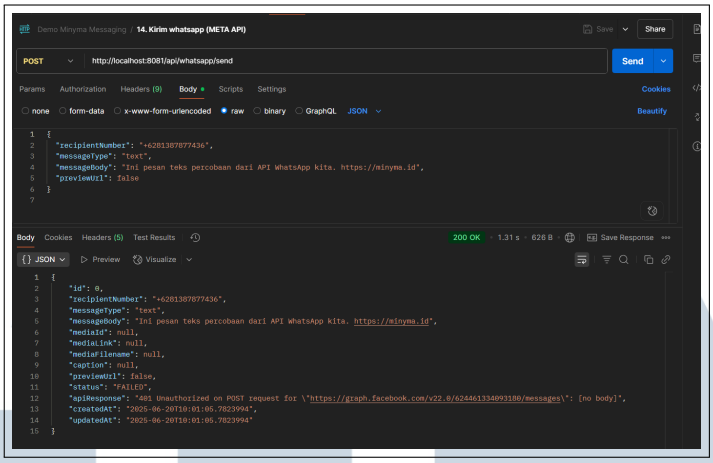
E Struktur *Package* WhatsApp

Struktur modul WhatsApp di dalam sistem *backend* dibagi ke dalam beberapa bagian berikut untuk mendukung pemisahan logika dan tanggung jawab:

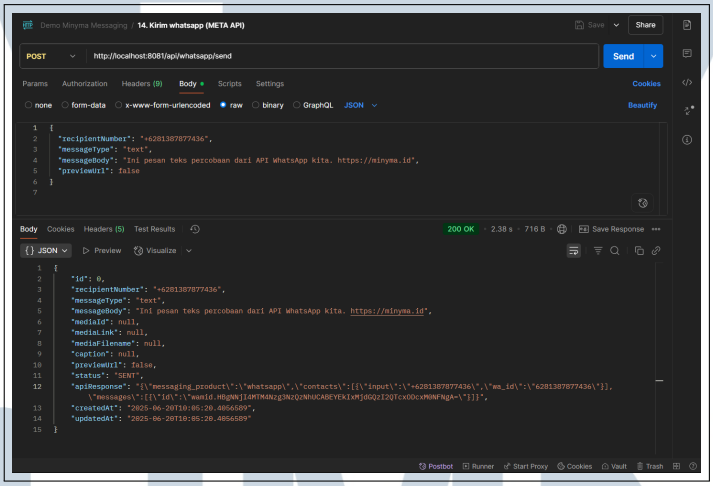
- *Controller*: Menyediakan *endpoint* REST untuk menerima permintaan pengiriman pesan WhatsApp dari pengguna dan meneruskannya ke layanan terkait.
- *Service*: Mengelola logika utama pengiriman pesan WhatsApp, termasuk validasi, pemanggilan API pihak ketiga, dan pencatatan status pengiriman.
- *DTO*: Menyediakan struktur data untuk memetakan permintaan pengguna, seperti nomor tujuan dan isi pesan, agar dapat diproses oleh sistem secara konsisten.
- *Model*: Merepresentasikan entitas pesan WhatsApp yang mencakup informasi seperti nomor penerima, isi pesan, waktu kirim, dan status pesan.
- *Repository*: Bertanggung jawab terhadap penyimpanan dan pengambilan data pesan dari basis data menggunakan pendekatan JDBC.

F Pengujian dan Validasi Pesan WhatsApp

Pengujian terhadap *endpoint POST /api/whatsapp/send* dilakukan menggunakan token sementara karena proses verifikasi bisnis melalui Meta API masih belum selesai sepenuhnya. Gambar 3.24 menunjukkan respons gagal akibat token yang telah kedaluwarsa, sedangkan Gambar 3.25 menunjukkan hasil ketika token aktif digunakan dan pesan berhasil dikirim ke nomor WhatsApp yang telah diverifikasi. Meskipun sistem sudah siap dan berjalan sesuai alur, keterbatasan pada tahap verifikasi membuat pengujian masih bersifat terbatas.



Gambar 3.24. *Token unauthorized* saat pengujian *endpoint* pengiriman WhatsApp



Gambar 3.25. Pesan berhasil terkirim ke nomor WhatsApp yang diverifikasi

3.4 Kendala dan Solusi yang Ditemukan

Selama proses pengembangan sistem *Messaging Service*, ditemukan berbagai kendala teknis maupun non-teknis yang dialami sebagai berikut.

1. Salah satu kendala masalah verifikasi akun bisnis di platform Meta Developer yang digunakan untuk integrasi layanan WhatsApp. Hingga akhir masa magang, akun bisnis belum diverifikasi oleh Meta, sehingga pengiriman pesan hanya dapat dilakukan ke nomor yang telah didaftarkan sebagai penguji. Kondisi ini menghambat implementasi fitur.

2. Saat melakukan pengiriman email secara massal, sistem mengalami *overload* karena semua permintaan diproses secara bersamaan tanpa mekanisme

antrean. Hal ini menyebabkan server menjadi tidak responsif dan banyak permintaan gagal terkirim. Tidak adanya mekanisme *retry* dan pembatasan *rate* terhadap API memperparah situasi.

3. Pola kerja daring selama magang menyebabkan miskomunikasi dalam memahami kebutuhan fitur dan pembagian tugas. Tidak adanya sistem manajemen proyek formal membuat progress antar anggota sering tidak sinkron dan menyebabkan keterlambatan dalam penggabungan modul.

Melalui kendala tersebut, berikut merupakan solusi yang dilakukan.

1. Untuk mengatasi masalah verifikasi akun WhatsApp, digunakan token sementara dari Meta Developer untuk uji coba terbatas. Dengan pendekatan ini, fitur dasar tetap dapat dikembangkan meskipun akses masih terbatas. Dokumentasi resmi dari Meta terus dipantau untuk mengetahui perkembangan status verifikasi.
2. Diterapkan *middleware* RabbitMQ sebagai antrean pesan agar proses pengiriman email berlangsung secara asinkron. Mekanisme *retry* ditambahkan untuk menangani kegagalan akibat *error* jaringan atau batasan kuota dari vendor API. Proses pengiriman juga dilengkapi dengan *logging* yang mendetail.
3. Digunakan WhatsApp sebagai alat komunikasi utama, ditambah Google Sheet sebagai alat pelaporan harian. Setiap anggota mencatat progress dan kendala yang dihadapi, sehingga koordinasi menjadi lebih teratur dan pengembangan sistem dapat dilakukan secara paralel tanpa hambatan besar.